University of Groningen

Programming Fundamentals

Arnold Meijster

Academic year: 2024-2025

Themis: Greek goddess of divine justice

Contents

Chapter 1

Introduction

1.1 Creating and running a C program: step-by-step

1.1.1 Using an editor

The source code of a C program is a plain text file with the file name extension `.c`, for example `myprogram.c`. Since the source code of a program is just an ordinary text file, you can create and edit it with any text editor that is installed on the lab systems. An editor is basically a simplified word processor that stores its files as simple ASCII text files (so, fonts and fontsizes are of no concern). Many students prefer to use the editor `geany`. Of course, you are free to use any other editor.

1.1.2 Compilation

Before you can run a program you must first convert its source code into a form that is executable by the computer. This conversion process is called *compilation*. The program that performs the conversion is called the *compiler*.

On a standard UNIX system the compiler is called `cc`. However, we will use a compiler named `gcc`. This compiler is a free (open source) alternative for the standard system compiler and there is a version available for almost every operating system (including Linux, MacOSX and all versions of Microsoft Windows). Therefore we prefer `gcc` over a platform specific compiler.

Over the course of time, multiple versions (dialects) of the programming language C have appeared. During the course Imperative Programming we use a specific variant, the so-called C99 version. To make sure that the compiler compiles this particular version, we need to invoke the compiler using the option `-std=c99` . Moreover, we want the compiler to produce feedback if it detects that we may have done something wrong. The compiler options `-Wall` and `-pedantic` produce warnings about everything that the compiler finds suspicious. So, we compile the program in the file `myprogram.c` using the command

```
gcc -std=c99 -Wall -pedantic myprogram.c
```

If the compilation succeeds, a file `a.out` is generated. This file, which is not readable by humans, contains machine code that can be executed by the processor of the computer. This file is therefore called the *executable*.

The compilation fails if the compiler detects an error in the source code. In this case the compiler will report an error message, along with the file name and line number where the error was detected. It is usually clear from this report where the error in the program is located. However, be warned that the error in the source code might actually be one line (or even a few lines) earlier than the line number reported by the compiler.

Tip!

The editor `geany` can show line numbers (see option "`Show line numbers`" in the "`View`" menu). This way you can quickly find the lines with the line numbers that are reported by the compiler. You can also run a program directly from within `geany`, so you do not need to switch continously between multiple windows.

Tip!

If you work on a larger programming project then you will probably split your program into several files with the file name extension `.c`. You can compile all these files at once using the command

```
gcc -std=c99 -Wall -pedantic *.c
```

1.1.3 Execution of a program

After compilation of the source code, the program is executed by simply typing:

```
./a.out
```

Tip!

Sometimes you want to test your program with a particular input. In that case it might be a good idea to type this input in a file, and use the content of this file as the input for your program. This saves you the burden of typing the input for each test that you perform. You can supply the executable with the input from this file using *input redirection*. For example, if the file name of the input file is `01.in`, then you use the following command to supply the input to `./a.out`:

```
./a.out < 01.in
```

Similarly, we can save the output of a program using *output redirection*. The output of the program is written to a file, so it will no longer appear on the screen. An example command-line is:

```
./a.out > 01.out
```

We can also combine input redirection and output redirection as you can see in the following example:

```
./a.out < 01.in > 01.out
```

1.2 Structure of a program

A C-program consists of a number of standard parts. It is quite cumbersome to type in these standard lines of code over and over again. It is therefore convenient to have a small skeleton program at hand. During this course we will use the following little skeleton program as a starting point for each program that we will make:

```
/* file : skeleton.c */
/* author : Joe Sixpack (joe@student.rug.nl) */
/* date : Mon Sep 3 2018 */
/* version: 1.0 */
/* Description:
 * Please type here a description of
 * what this program is supposed to do.
 */

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

int main(int argc, char *argv[]) {
  /* here you place your own code */
  return 0;
}
```

At this time, this code may be completely incomprehensible to you. Do not worry, you will understand its structure very soon. However, now is a good moment to create a file `skeleton.c` with this program fragment, so that you do not have to type this code whenever you want to start a new programming project.

1.2.1 Statements

The core of a program consists of a sequence of *statements*. A statement is an elementary command, such as "increase the value of the variable `x` by 3" or "repeat the following 10 times". Usually, each statement is placed on a separate line and ends with a ; (semicolon). The computer executes statements one-by-one and in the order in which they appear in the program.

1.2.2 Comments

A *comment* in a C-program is a human-readable text that is enclosed by /* and */. Note that you can type comments that consist of one or multiple lines as you can see in the code of the skeleton program. Comments are ignored by the C compiler and are intended to make (the logic of) the program easier to understand by yourself and other people.

The skeleton program shows a good convention: it is customary to place at the very beginning of a program comments that contain information about who wrote the program and when. It also contains a description of the functionality of the program and possibly auxiliary information.

Since the compiler ignores comments, many programmers have become lazy and hardly use comments in their programs. This is really bad practice! You can help others to understand the logic of your code by giving helpful comments, so please use this opportunity. Besides, you also benefit yourself. For example, if you want to add some functionality to a program that you made a long time ago, it is very unlikely that you will still remember all the details of the design of the program. In this situation good comments and documentation are very helpful.

1.2.3 Indentation and layout convention

With *white-space* we mean spaces, tabs and newlines. A newline, also known as a line break or end-of-line (EOL) marker, is a special character signifying the end of a line of text. In the programming language C, the programmer is free to use white-space whenever (s)he likes. White-space is used to get an acceptable layout (indentation) of a program and is completely ignored by the compiler. Be warned that there are programming languages that do enforce restrictions on indentation, for example the programming language Python. Since the C-compiler ignores white-space, the following two programs are completely equivalent:

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

```
#include <math.h>

int main(int argc, char *argv[]) {
  printf("Hello world\n");
  return 0;
}
```

```
#include <stdio.h> #include <stdlib.h>
#include <math.h>
      int
main (int argc, char *
argv[ ] ) { printf ("Hello world\n") ;

       return
0; }
```

Clearly, the first program is preferred since it is much easier to read. Since white-space is ignored by the compiler, we can use it to make our program more readable. In C, pieces of code are embraced by the braces { and }. These pieces of code, including the braces, are called *blocks*. The code of a block without the braces is called the *body* of a block. During this course, we adopt the convention to indent the body using some extra white-space (2 spaces is a nice default). Since blocks can be nested (a block within another block), this automatically yields levels of indentation. We adopt the convention that an opening brace { is directly followed by a newline, and we increment the indentation level for the body of the corresponding body. We decrement the identation level when we arrive at the closing }. Note that we place the closing } an indentation level lower than the body of the corresponding block. An example of this coding convention is given in the following program:

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

/* this is indentation level 0 */
int main(int argc, char *argv[]) {
  /* this is indentation level 1 */
  if (1 < 2) {
    /* this is indentation level 2 */
    printf("Ok\n");
  } else {
    /* this is also indentation level 2 */
    printf "Houston, We've a problem\n");
  }
  /* We are back at indentation level 1 */
  printf("Goodbye\n");
  return 0;
}
```

1.3 Variables

In imperative programming languages, data is stored and modified in *variables*[1]. Data can be anything, for example numbers or text. In imperative languages variables play an important role,

[1]This is in contrast with so-called *functional programming languages* in which variables are bound to expressions and keep a single value during their entire lifetime (they are not modifiable). A discussion about the pros and cons of imperative programming versus functional programming is outside the scope of this course.

so this also applies to C. You can imagine a variable as a 'drawer' with a label (the name of the variable) on it, in which you can store an item (value). At any point in your program each variable has ('contains') a certain value, the 'content' of the 'drawer'.

Alert!

A variable in a programming language is different from a variable in the mathematical sense. Sometimes, mathematical variables are called 'unknowns', which is actually a much better term, since the value is usually not known. At any time during the execution (but *not* at the time of writing) of the program, the value of a variable is known.

Each variable has a *type*. The type of a variable determines what kind of value you can store in it. For example there are numeric types (for numbers), and character types (for texts).

1.3.1 Identifiers

An *identifier* is a name for a variable or function. Identifiers begin with a letter. The remainder of the name can consist of letters, numbers and underscores.

Alert!

Upper case and lower case letters are important! For example, the identifiers `value` and `Value` are different! We say that the programming language C is *case-sensitive*. In this course, we adopt the following convention. An identifier starts with a lower case letter. In the rest of the identifier, each new word starts with a capital letter while the other letters are in lower case. Examples of identifiers using this format are `thisIsAnExample`, `notEqualTo`, and `hasValue1`. In the C programming literature you will find many conventions for the format of identifiers. The one that we will use is called *camelCase*.

1.3.2 Declaration

Before you can use a variable, you need to tell the C compiler about the existence of the variable. You introduce a variable by designating its type followed by the name of the variable (identifier) and a semicolon. This is called the *declaration* of the variable. Here is an example of three declarations:

```
double cheeseburger;
char broiled;
long timeAgo;
```

After these declarations, the variables cheeseburger, broiled and timeAgo exist and can be used in the remainder of the program. A discussion of the available types can be found in Section 1.4.

Before you inspect the value of a variable, it must first contain a value. We will use the terms *reading* a variable for inspecting the content of a variable and *writing* a variable for modifying the content of a variable. The process of writing an initial value to a variable is called the *initialization* of the variable. A variable can be declared and supplied with an initial value in one single statement, as is shown in the following example:

```
double pi = 3.14159265;
int two = 2;
```

It is good practice to add comment lines to your program that describe what the main variables of your program are used for.

1.3.3 Assignments

An *assignment* is used to modify the value of a variable. An assignment is a statement that has the form <variable> = <expression>;

```
answer = 17 + 25;
```

In this example, the value of the expression 17 + 25 will be calculated and assigned to the variable answer. Note that the use of the symbol = is a bit misleading: it is not used to denote mathematical equality! It would actually be better to use the definition symbol := (which is used in the programming language Pascal and its derivatives) or an arrow, <− , but unfortunately the designers of the C programming language decided otherwise.

In the above example, the right hand side of the assignment is a constant expression (it evaluates to the constant 42). However, the right hand side need not be constant and can be an expression containing variables. An example of such an assignment is:

```
c = 2*a + b;
```

In this example, the value of the variable a is multiplied by two and the value of the variable b is added. The resulting value is stored in the variable c.

Often, the assignment is an operation on the variable on the left hand side of the equality symbol. A typical example would be a statement that adds 5 to the value of a variable i. To do this, the 'normal' statement would be i = i + 5. However, the special assignment operator += exists for this particular assignment:

```
i += 5;
```

Likewise, we have −=, *= and /=.

It is very common to use a variable as a simple counter that needs to be incremented or decremented regularly. Of course, we can increment a variable counter by one with the statement counter = counter + 1 or counter += 1. However, there exists another convenient shorthand notation for this assignment. This notation is of the form <variable>++;. The operator ++ is called the *increment operator*. Thus, the statement counter = counter + 1 can be implemented as:

```
counter++;
```

Likewise, we have the *decrement operator* −−:

```
counter−−;
```

There exists two versions of the increment and decrement operators. The increment version explained above is called the *post-increment operator*, since the symbol ++ appears directly after the variable counter. The other version is the *pre-increment operator* in which the symbol ++ appears directly in front of the variable. An example of this would be

```
++counter;
```

The difference between the two versions is subtle. The pre-increment and pre-decrement operators increment (or decrement) their operand by 1, and the value of the expression is the resulting incremented (or decremented) value. In contrast, the post-increment and post-decrement operators increase (or decrease) the value of their operand by 1, but the value of the expression is the operand's original value prior to the increment (or decrement) operation. This is clarified in the following example:

```
a = 40;
b = a++;
c = ++a;
```

After execution of this code fragment, we have a=42, b=40 and c=42. Note that the second and the third line of this example are in fact two assignments in one single statement! The code of the above program fragment is equivalent to the following code:

```
a = 40;
/* b = a++ is equivalent to */
b = a;
a = a + 1;
/* c = ++a is equivalent to */
a = a + 1;
c = a;
```

Since the increment/decrement operator modifies its operand, use of such an operand more than once within the same expression can produce undefined results. For example, in statements such as x = (x−−)*(++x), it is not clear in what order the decrement and increment operators are performed. Therefore, never use ambiguous constructs like these!

1.4 Types

1.4.1 Integers

An *integer* is a number without a fractional component. For example, 21, 4, 0 and -2048 are integers, while 2.5 is not an integer. The programming language C has different types for the storage of integer numbers. These types differ in the maximum (and minimum) value that they can attain. Integer numbers are represented in a computer by a series of ones and zeros (called *bits*). A bit can have only one out of two values: zero or one. A sequence of n bits in a row can be used to encode each unsigned (non-negative) integer number less than 2^n uniquely. This representation is called the *binary representation* of the integer number. In principle, this representation works the same as the decimal representation of a number. For example, with the decimal number 135 we encode in fact

$$135 = 1 \cdot 100 + 3 \cdot 10 + 5 = 1 \cdot 10^2 + 3 \cdot 10^1 + 5 \cdot 10^0.$$

Likewise, the (16 bit) binary representation of 135 is 0000000010000111, since

$$135 = 128 + 4 + 2 + 1 = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0.$$

Of course, with a sequence of n bits we can only represent 2^n integer numbers (the numbers 0 through $2^n - 1$). Hence, for numbers equal to or bigger than 2^{16} we need more than 16 bits. In ANSI C, there are several standard types available for the representation of positive integers, namely the types **unsigned short**, **unsigned int** and **unsigned long**. For each type, a fixed number of bits is reserved. However, the number of bits per type may be different on different computer architectures. On a standard PC using `gcc`, the following values apply:

type	bits	maximum value
unsigned short	16	65535
unsigned int	32	4294967295
unsigned long	64	18446744073709551615

In order to represent negative numbers as well, we need to incorporate the minus sign in the binary representation of a number. Several representations exist, but they all need one bit for the sign. Effectively, this halves the maximum value that we can represent for each integer type. For example, with n bits we can represent signed integer numbers from -2^{n-1} to $2^{n-1} - 1$. We arrive at the following table:

type	bits	minimum	maximum
short	16	-32768	32767
int	32	-2147483648	2147483647
long	64	-9223372036854775808	9223372036854775807

For most practical purposes, the type **int** provides enough bits to represent the integers that we need in our programs without needing to worry about whether the number is signed or unsigned. Therefore, we will use it as the default type for storing integers.

Alert!

If you want to use a constant decimal integer number in your program, you can simply use its decimal notation, e.g. x=42. However, do not start decimal numbers within your program with a leading 0! Obviously, zero itself is an exception. Numbers that start with 0 are interpreted as so-called *octal* numbers: 042 is then interpreted as 34 (i.e. 4*8+2). So, the assignment x=042 is equivalent to the assignment x=34. Besides octal numbers, also hexadecimal numbers exist. During this course, we will only make use of decimal numbers.

1.4.2 Floating point numbers

Of course, we sometimes need non-integer numbers like 3.14159. These numbers are called *floating point numbers* (or *fractional numbers*).

The types **float**, **double** and **long double** are used for storing floating point numbers. How these numbers are represented in a binary format is dependent on the compiler and computer that we use. The only thing that is specified is that a **double** has at least as much precision (and range) as a **float** and a **long double** at least as much precision (and range) as a **double**. Which one to choose depends on the application that you are programming.

Note that floating point values in your program code are written using a dot (.), e.g. 3.14 (do not use 3,14). Moreover, with e one can indicate exponents in scientific notation. For example, 2.3e−2 represents 2.3×10^{-2} or 0.023.

On a standard PC the type **long double** has the same range and precision as the type **double**. For a standard PC we find the following ranges:

type	bits	maximum
float	32	$\pm 10^{38}$
double	64	$\pm 10^{308}$
long double	64	$\pm 10^{308}$

1.4.3 Characters and text

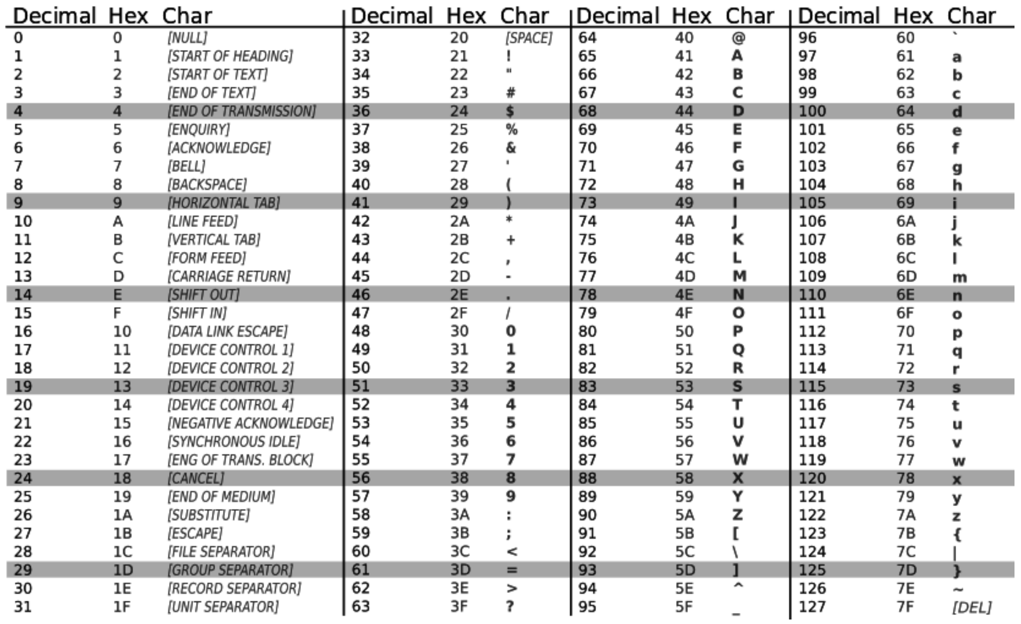
A single letter is called a *character*. Examples of characters are: 'a', 'A', 'z', 'Z', '6', '+', and '-'. A character has the type **char**. Characters in the source code of a program are enclosed by *single quotes*:

```
char first = 'A';
char last = 'Z';
```

Alert!

Characters are represented as (small) integers. The type is wide enough to represent each character as a positive value. About the type **char** there is a persistent misunderstanding. It is not true that a **char** is always represented by one byte (8 bits). Although this is often the case, it is by no means required by the C-language definition. However, it is defined that a **char** is at least 7 bits (not 8!) wide.

Because characters are represented as numbers, we can apply arithmetic operations on them. For example, the expression '1'+ '2' is perfectly valid. The result itself is a char, so it is not the text "21" nor the character '3'. The correct result is obtained by adding the numerical values 49



Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Figure 1.1: The ASCII table: ASCII=American Standard Code for Information Interchange

and 50 (the locations of '1' and '2' in the ASCII table of characters), so we get the character at location 99 in the ASCII table, which is the character 'c'.

As you can see in Figure 1.1, the characters 'A', 'B', .., 'Z' are a consecutive subsequence of the ASCII table. So, the expression 'A'+ 1 is a complicated way to write 'B'. The same holds for the letters 'a', 'b', .., 'z' and the digits '0', '1', .., '9'. So, the expression '5'+ 1 yields the character '6'.

A sequence of characters (text) is called a *string*. Strings in the program text are enclosed by *double quotes*. Often, we wish to include *non-printable* characters in a string. This can be done using special representations for these characters. For example, a line break in a string is represented by \n . The notation that uses a backslash (\) in front of a printable character is called *escaping*. For example, the program fragment (see also section 1.5)

```
printf("The quick brown fox\njumps over the lazy dog.\n");
```

will result in the output:

```
The quick brown fox
jumps over the lazy dog.
```

In the same way, \t indicates a tab character. Double quotes in a string (normally this would indicate the end of the string) can also be escaped, so we can use the notation \" to include quotes in a string. The backslash is also escaped, so we get \\ . A somewhat more complicated example is:

```
printf(" 3\t4\t5\t\\12\n 4\t5\t6\t\\\"15\"\n");
```

This code fragment will produce the following output:

```
 3       4       5       \12
 4       5       6       \"15"
```

1.5 Output

In the C programming language the output functions (functions used to write to the screen or a file) are not part of the language itself. As a result, a programmer can write his/her own output functions and tailor them to his/her particular needs. However, this is quite a lot of work. For this reason, there is a set of basic output functions (as well as many other auxiliary functions) already available. This set of functions is called the *standard C-library* or *libc*. You can use these functions after you imported them. This is actually the reason why our programs start with:

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

The 'name' `stdio.h` refers to the Standard Input and Output *header file*. In this file (on a Unix system you will probably find this file at the location `/usr/include/stdio.h`) all functions that are related to input and output are defined.

1.5.1 printf

The output of a C program goes to a file called *stdout*, the *standard output* (usually the screen). The function printf() is used for this. It is probably the most frequently used function from the C library. In fact, we already encountered this function a few times. The function can be used as follows:

```
printf("Hello ");
printf("my friend,\n");
printf("how are are you?\n");
```

This code fragment will result in the following output:

```
Hello my friend,
how are you?
```

Apparently, text to be printed is enclosed by double quotes. Note that \n is used to denote the line break (or newline character).

The above example is only one of the many uses of printf. With printf it is possible to format the output: you can choose the layout of the output using a so-called *format specification*. The general use of printf is as follows:

```
printf("format specification", ...);
```

The first argument of the function printf is the format specification, which is a string, and must therefore be enclosed by double quotes. The remaining arguments (if any) are a list of values to be printed according to the format specified. A complete description of all possible format specifications is beyond the scope of these lecture notes. For the course *Imperative Programming* it suffices to know the following format specifiers:

- %d : print an integer value.
- %ld: print a long integer value.
- %u : print an unsigned integer value.
- %lu : print a long unsigned integer value.
- %f : print a float value.
- %lf : print a double value.
- %c : print a character value.

- %s : print a string.

Furthermore, there are a few special symbols that can be used to format the output:

- \n : line break (newline character).
- \r : carriage return (cursor goes to beginning of current line).
- \t : tab.
- \b : backspace.
- \l : formfeed (for printers).
- \\ : print a backslash (\).
- \" : print a double quote (").

Here are some examples that demonstrate the usage of printf:

```
printf("If we multiply %d and %d, we get %d.\n", a, b, a*b);
printf("His name is %s.\n", "John");
printf("The value of pi is %f, and the value of %c is %f\n", 3.14159, 'e', 2.71828);
printf("The C-notation for a backslash (i.e. \"%c\") is \\\\\n", '\\');
```

1.6 Expressions

An *expression* is a formula in your program that evaluates to a value of some type. As you will see, this is a very broad concept. These are all examples of expressions.

```
5
a
b + 1
"Hello"
b * (7 + a)
'A' + 1
a > b
```

Let us assume that the variables a and b are of the the type **int**. The name of a variable is itself an expression that evaluates to the value of that variable. So, b + 1 is the expression that yields the value of b incremented by one (which clearly also has the type **int**).

We can, just like in ordinary calculus, change the order in which expressions are evaluated using parentheses. The operator-precedence rules are mostly what you are used to: for example, the expression b*7+a means (b*7)+a instead of b*(7+a).

Each expression yields a value of a specific type. For example, the expression 2 + 4 yields a value of the type **int**, while 'A'+1 yields a character (**char**). A special case is the expression a > b. This is called a *boolean expression*, which is either true or false. In many programming languages (such as Java, Pascal, C++) there is a special data type reserved for boolean expressions. In C there is no such data type. Instead, any integer type can be used to represent boolean values, using the following interpretation: the value zero represents the boolean value *false*, while any non-zero value is interpreted as *true*. The expression 5 <2 is false and therefore evaluates to the value zero. The code below thus initialises, in a complicated way, the variable zero to 0:

```
int zero = 5 < 2;
```

1.6.1 Arithmetic operators

The basic arithmetic operations addition, subtraction, multiplication and division can be translated directly into C-operators. Addition and subtraction use simply $+$ and $-$, multiplication is done with $*$ and division with $/$. The operator precedence rules are standard: multiplication and division have higher priority than addition and subtraction. So, 5*8+2 evaluates to 42 (and not to 50=5*(8+2)).

Alert!

If both operands of the operator / are **int**-values, then the resulting value will also be an **int**-value. This case is known as *integer division*: if both operands are positive **int**-values, then the digits after the decimal point are truncated. For example, 5 / 2 evaluates to 2 (and not 2.5). Integer division truncates, so it does not round: the value of 999/1000 is therefore 0! If at least one of the operands of the operator / is a floating-point value, then the resulting value is also a floating point value: the value of 5.0 / 2 is 2.5 (as you expect). In fact, the expressions 5 / 2.0 and 5.0 / 2.0 evaluate to 2.5 as well.

Besides integer division, there is the *modulo operator* %, which yields the remainder after division. For example, 17%5 yields the value 2 (since 17 equals 3*5+2). Note that both operands of the modulo operator need to be of the type **int**.

Alert!

Be warned that the value of a % b need not be in the interval [0,b). If a is less than zero, then a % b will evaluate to a negative value as well. For example, the expression −17%5 yields −2 (since −17==−3*5 − 2). The following identity always holds for integers x and y (except for y==0):

```
(x / y) * y + x % y == x
```

1.6.2 Math operators

Besides the basic arithmetic operators, C also features most of the more advanced math operations that can be found on comprehensive calculators. If we want to use these operations, we need to include the file `math.h`:

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

In the file `math.h` many mathematical functions are already pre-defined. During the course Imperative Programming we will not make extensive use of these functions, but it is good to know that they are available. Some of these functions are given in the following table:

function	C function
$\sqrt{x}$	sqrt(x)
$\lvert x\rvert$ (type **int**)	abs(x)
$\lvert x\rvert$ (type **double** of **float**)	fabs(x)
$\sin(x)$, $\cos(x)$, $\tan(x)$	sin(x), cos(x), tan(x)
$\sin^{-1}(x)$, $\cos^{-1}(x)$, $\tan^{-1}(x)$	asin(x), acos(x), atan(x)
e^x	exp(x)
x^y	pow(x, y)
$\ln(x)$	log(x)
$\log_2(x)$	log2(x)

Alert!

If you use functions from the include file `math.h`, then you must append −lm to the standard compile command. This has the effect that the math functions that you have used from the standard math-library get included in the file a.out.

```
gcc −std=c99 −Wall −pedantic myprogram.c −lm
```

In fact, it is never wrong to use this compile command, even if you did not use any math functions.

1.6.3 Coercion

If you multiply two **int**s, the result will also be an **int**. But, what is the resulting type of the multiplication of a **double** and an **int**? In this case, the compiler automatically converts the smaller type (here **int**) into a value of the larger type (here **double**), and then the multiplication is executed. The resulting value is the largest of the two types, so in this case it would be a **double**.

This implicit *type conversion* is known as *coercion*. The automatic type conversion is done by the compiler (at compile time). In mixed-type expressions, one type must be the largest (i.e. have the largest binary representation), which is the 'supertype' of this expression. In such expressions, the smaller subtypes are implicitly converted (promoted) to the supertype. The expression is evaluated, and the result is a value of the supertype.

For example, consider the following code snippet:

```
short a = 1;
int b = 2;
long c = 3;
c = (a + b) + c;
```

Of course, the final value of c is 6, but it is interesting to see how the expression (a + b)+ c is evaluated. First, since it is between parentheses, the expression a + b is evaluated. Because a is a **short** and b an **int**, the value of a is implicitly converted (coerced) to an **int** and then added to the value of b. The result is therefore an **int** value. This value, on its turn, is coerced into the larger **long** type and added to c, resulting in the **long** value 6. This value is assigned to c.

Integer types are coerced from small to large types, i.e. in the direction **char** → **short**→ **int**→ **long**. Such an integer conversion is always safe, i.e. no overflow/underflow errors can occur. The same is true for conversion of floating point values. They are coerced in the direction **float** → **double**.

However, if we try to mix integer and floating point types in a single expression, then weird errors may occur. For example, it is defined that an **int** is coerced automatically into a **float** (although they are both 32 bits on a PC) or a **double**. Due to the granularity of the representation of floating point numbers, this may result in an unsafe conversion simply because some exact **int** values cannot be represented as an exact **float** value.

Alert!

Consider the following declaration and assignment:

```
double x;
x = 3;
```

Maybe you did not spot it, but in this assignment coercion takes place: the **int** value 3 is coerced to the **double**-value 3.0 since the left-hand side of the assignment is a **double**.

Alert!

Consider the following code snippet:

```
short a = 50;
int b = 100;
double x;
x = a / b;
```

You might hope that the final value of x is 0.5, but in fact it is 0.0 since the division a/b results in the **int** value 0 (integer division truncates). Next, the **int** value 0 is coerced to the double value 0.0 and assigned to x.

1.6.4 Casting

Casting (in full *type casting*) is a way to *explicitly* convert the type of the result of an expression into some other type. The notation that is used for this conversion is:

```
(<type>)expression
```

For example, with (**int**)3.7 we convert the **double**-value 3.7 into the **int**-value 3. Note that floating-point values that are converted to integer values are not rounded: the digits after the decimal point are truncated.

Of course, if you cast a larger type into a smaller type, you will lose precision. For example, if you cast an **int**-value (which is 32 bits on a PC) into a **short**-value (which is 16 bits), then you loose bits. This means that, on a PC, an **int** value that needs more than 16 bits to be represented exactly, will not 'survive' the cast to a **short**. The result is not defined, but is very likely the **short**-value that is represented by the last 16 bits of the original **int** value.

Alert!

The type cast operator has a high priority. For example, (**int**)3.5 ∗ 2 evaluates to 6 since 3.5 is converted to an **int** (hence 3) before the multiplication. So, we need to write (**int**)(3.5 ∗ 2) to get the value 7.

Alert!

The addition of an **int** and a **char** gets automatically converted into an **int**. It is perfectly acceptable to do this. However, if the result of this addition is assigned to a **char**-variable, it could very well be that the result is larger than the largest value that is available in the character set. This is called an *overflow*. Printing an overflowed character value may produce funny characters on the screen.

1.6.5 Relational operators

Often, we need to compare the values of two expressions. We have six *relational operators* (or *comparison operators*) at our disposal:

operator in C	<	<=	==	>=	>	!=
mathematical meaning	$<$	$\leq$	$=$	$\geq$	$>$	$\neq$

The result of a comparison (i.e. the value of the expression) is either 0 (in case the comparison is false) or 1 (in case the comparison is true). Note that a comparison never returns another positive number (like 2, or 42), even though all positive integers are considered to represent the boolean value *true*.

Alert!

Often, the assignment operator = is accidentally written where the comparison operator == is meant. This error is often not detected by the compiler, because an assignment itself is a valid expression that yields the assigned value (as a side-efffect). So, `a == 3` yields a boolean value indicating whether `a` has the value 3, whereas the expression `a = 3` assigns the value 3 to `a` and also returns the value 3! The assignment `y = 3 * (x = 3)` is therefore a tricky (and ugly) way to assign to `x` the value 3 and to `y` the value 9 in one statement. Similarly, `y = 3 * (x == 3)` is a tricky way to assign to `y` the value 3 in the case that `x` equals 3, and zero otherwise.

1.6.6 Boolean operators

We already know that for boolean expressions, which evaluate to either *true* or *false*, integers are used with the interpretation that zero is used to represent *false*, while any non-zero value is used to represent *true*.

The *boolean operators* are operators that have boolean expressions as operands and also return boolean values. These operators stem from Boolean algebra (logic) and are summarized in the following table.

operation	pronounciation	meaning
`a && b`	"a and b"	true (1) if both `a` and `b` are true (1), otherwise false (0)
`a || b`	"a or b"	true (1) if `a`, `b` or both are true (1), otherwise false (0)
`!a`	"not a"	true (1) if `a` is false(0), otherwise false (0)
`a ^ b`	"a xor b"	true (1) if either `a` or `b` is true (1) but not both, otherwise false (0)

Here are some examples of the use of these operators:

```
readyToGo = doorIsLocked && !lightsOn;
canBuy = (hasCash && money >= price) || hasCreditCard;
```

The relative priority of these operators is (from high to low) !, ^, &&, ||. For example, the boolean expression `1 || 1 && 0 || 0` evaluates to true (1), since it is evaluated as `1 || (1 && 0)|| 0` (and not `(1 || 1)&& (0 || 0)` which would yield false).
For readability, it is often better to use explicit parentheses even though the priority rules allow you to omit them.

Evaluation of boolean operators with two arguments is performed in a *short-circuit* fashion: if the outcome of the operator is already known due to the value of the first operand, then the value of the second operand is not computed. For example, it is safe (but strange) to write `(1 == 2)&& (1/0 == 3)`. Since `(1 == 2)` already evaluates to `0` (false), the result of the operator `&&` will be `0`. Hence, the expression `1/0 == 3` is not evaluated at all and a "`division by zero`" error will not occur. The expression `(1/0 == 3)&& (1 == 2)` however is logically (mathematically) equivalent, but will result in a "`division by zero`" error. Especially in conditional statements (see Section 1.7) we will use this feature quite often.

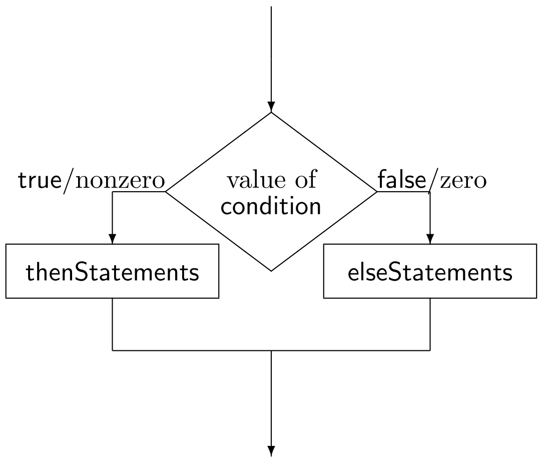


Figure 1.2: schematic representation of the **if**-**else**-statement

1.7 If-statement

The syntax of the **if**-statement is as follows:

```
if (condition) {
    thenStatements;
} else {
    elseStatements;
}
```

The condition must be a boolean expression and therefore needs to evaluate to a non-zero (true) or a zero (false) numeric value. In the first case, the statements of the then branch are executed. In the latter case, the statements of the **else** branch are executed. This is schematically shown in Figure 1.2.
An example:

```
if (x % 2 == 0) {
    /* x is even: divide by 2 */
    x /= 2;
} else {
    /* x is odd: multiply by 3 and add 1 */
    x = 3*x + 1;
}
```

Often, there is no need to have an **else**-branch, in which case it can be omitted:

```
if (condition) {
    thenStatements;
}
```

An example without an **else**-branch:

```
/* x == A, y == B */
if (x < y) {
    x = y;
}
/* x == max(A,B) */
```

1.8 Conditional expression

In C, there is a shorthand assignment to implement the following **if**-statement:

```
if (n >= 0) {
  absValue = n;
} else {
  absValue = −n;
}
```

The shorthand assignment is

```
absValue = (n >= 0 ? n : −n);
```

The right-hand side of this assignment is called the *conditional operator* or *ternary operator*. The obscure name 'ternary operator' comes from the fact that the conditional operator is neither unary nor binary; it takes three operands.

The syntax of the conditional operator is

```
    e0 ? e1 : e2
```

and what happens is that the boolean expression e0 is evaluated, and if it is true then e1 is evaluated and becomes the result of the expression, otherwise e2 is evaluated and becomes the result of the expression.

In other words, the conditional expression is a kind of **if−else**-statement. The conditional operator, however, forms an expression and can therefore be used wherever an expression can be used (for example as the right-hand side of an assignment).

1.9 Switch-statement

The **if−else** statement is limited by the fact that only two cases can be distinguished. Sometimes, it is necessary to make more than two distinctions. You can solve this using nested **if−else** statements as in the following example:

```
if (grade == 10) {
  printf("outstanding\n");
} else {
  if (grade == 9) {
    printf("excellent\n");
  } else {
    if (grade == 8) {
      printf("very good\n");
    } else {
      if (grade == 7) {
        printf("good\n");
      } else {
        if (grade == 6) {
          printf("satisfactory\n");
        } else {
          if (grade == 5) {
            printf("almost satisfactory\n");
          } else {
            printf("unsatisfactory\n");
          }
        }
      }
    }
  }
}
```

The example above consists of 6 similar tests. Multiple nesting of the **if—else** statements is quite cumbersome, yields an ugly program layout and makes the program harder to understand. Therefore, it is better to use the **switch**-statement. In a **switch**-statement you can compare the value of the variable grade with constant values without nesting. If grade equals one of the listed values, the **case**s, then the corresponding statements are executed. So, we can rewrite the above example as follows:

```
switch (grade) {
case 10: printf("outstanding\n");
  break;
case 9: printf("excellent\n");
  break;
case 8: printf("very good\n");
  break;
case 7: printf("good\n");
  break;
case 6: printf("satisfactory\n");
  break;
case 5: printf("almost satisfactory\n");
  break;
default: printf("unsatisfactory\n");
}
```

When the execution of the program arrives at the **break** statement, it will continue execution at the closing brace of the switch statement. If the value of grade is not equal to one of the **case**s, then the statements of the 'case' **default** are executed.

Be wary of the feature of the **switch**-statement that is called *fallthrough*. A fallthrough happens when execution has reached the end of the code of a matching **case** that does not end with a **break** statement. In that case, the execution continues ("falls through") to the statements associated with the next **case**. This feature is used in the following example (note the grades 7 and 8, and 5 and 6):

```
switch (grade) {
case 10: printf("outstanding\n");
  break;
case 9: printf("excellent\n");
  break;
case 8: printf("very ");
case 7: printf("good\n");
  break;
case 5: printf("almost ");
case 6: printf("satisfactory\n");
  break;
default: printf("unsatisfactory\n");
}
```

1.10 Input: scanf

Besides output to the screen we also want to make use of keyboard input. With the function scanf we can read input from the keyboard in a similar fashion as printf writes output to the screen. The functions scanf and printf are so-called *I/O functions* or *I/O operations*, which is an abbreviation of Input/Output functions. In order to be able to use I/O functions, our program (as we already know) needs to start with

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

The general use of scanf is as follows:

```
scanf("format specification", ...);
```

The format specification is the same as in the function printf. However, the list with the remaining arguments is slightly different. The following example shows the difference between the argument list of scanf and printf:

```
int x, y;

printf("Type two integers: ");
scanf("%d %d", &x, &y); /* read x and y from the input (keyboard) */
printf ("The product of %d and %d is %d\n", x, y, x*y);
```

As you can see, the variables x and y in the argument list of scanf are preceded by an ampersand (&). At this moment, you do not need to understand why these ampersands are necessary. Later, in Chapter 3, you will learn what the ampersands exactly mean. For now it is sufficient that you know that you need to precede variable names with ampersands when you use the function scanf. If you forget the ampersands, then your program will most likely crash.

1.11 Tutorial exercises

Preliminary remark: in comments (between /* and */) there are assertions about the state of the variables. The assertion about the state before the assignment(s) is called the *precondition*. The assertion about the state after execution of the assignment(s) is called the *postcondition*. Be reminded that the = sign is used for assignment and does not mean equality. In the assertions (pre-and post-condition) equality is denoted by the == sign.

1.11.1 Compound assignments

The variables x, y, z are declared as follows:

```
int x, y, z;
```

Calculate in each of the following cases the post-condition in the given form.

1.	/*x == A, y == B */	x = x − y; y = y − x;	/*x == ?, y == ? */
2.	/*x == A, y == B */	x = x + y; y = x − y; x = x − y;	/*x == ?, y == ? */
3.	/*x + y == A, x + z == B */	x = x + y; y = y − z;	/*x == ?, y == ? */
4.	/*3x + 9y == A */	x = x + y; y = 2*y;	/*?x + ?y == A */

1.11.2 Derivation of (compound) statements

The variables x, y, z are declared as follows:

```
int x, y, z;
```

Find for each of the following cases a (compound) statement that realises the given postcondition starting from an initial state in which the precondition holds.

1.	/*x == A, y == B */	?	/*x == A + B, y == A − B */
2.	/*x == A, y == B, z == C */	?	/*x == B, y == C, z == A */
3.	/*3x + 8y == A */	?	/*x + y == A */
4.	/*x == A, y == B, z == C */	?	/*x == y, x + y + z == A + B + C */
5.	/*x == A, y == B */	?	/*x == max(A,B), z == min(A,B)*/
6.	/*x == A, y == B, A > 0, B > 0 */	?	/*x * y < 0, x + y == A + B */
7.	/*x == A */	?	/*A == 5 * y + z, 0≤z < 5 */

1.11.3 Integer division and remainders

The variables x, y, z are declared as follows:

```
int x, y, z;
```

Find for each of the following claims an equivalent expression in C:

1. x is odd.

2. x is a divisor of y.

3. x is a divisor of y and y is a divisor of z.

4. Either x is a divisor of y or y is a divisor of z, but not both.

5. The last digit of x (in decimal notation) is 7.

6. The last two digits of x (in decimal notation) are both 7.

1.11.4 Packing

Write a small program that asks the user to type in the length of the side of a cube, and the dimensions of a box, and prints the maximum number of cubes that can be stacked in the box. Note that the sides of each cube should be parallel to the sides of the box. You may assume that all lengths are integers (in centimeters).

1.11.5 Rounding

In many supermarkets prices are rounded to 5 cents. Write a program that asks the user to type in the number of cents that have to be paid, and print it rounded to 5 cents.

1.11.6 Pythagoras

Write a program that asks the user to type the lengths of three sides of a triangle and then prints whether this is a right-angled triangle or not. You may assume that the side lengths are integers.

1.11.7 Overlap

You can represent a circle by the coordinates of its center and its radius. In this exercise, you may assume that the coordinates and the radius are integers. Clearly, the radius is non-negative.

Write a program that reads in the data of two circles and then prints whether these two circles (partly) overlap each other or not. Note that two touching circles are considered to be overlapping.

Chapter 2

Loops

2.1 Iteration

In the programming language C there are three constructs available for *iteration* (or repetition): the **for**-statement, the **while**-statement and the **do while**-statement. The last two are intended for iterations where the number of steps is not known in advance. The **for** statement is intended for loops where the number of steps is known in advance.

A repetition construct is also called a *loop*. Each type of loop has a *body* and a *guard*. The body consists of the statements to be repeated. The guard is a boolean expression that determines how many times the body should be executed.

2.1.1 The while-statement

The syntax of the **while**-statement is as follows:

```
while (guard) {
    body;
}
```

The statements of the body are repeated as long as the guard evaluates to true. A schematic representation of this is shown in Figure 2.1. Usually, the statements of the body modify the variables in the guard such that eventually the guard evaluates to false and the loop terminates. As you can see in the figure, the guard is tested before the first statement of the body is executed. Thus, if the guard evaluates to false before the loop is started, the body will never be executed at all.

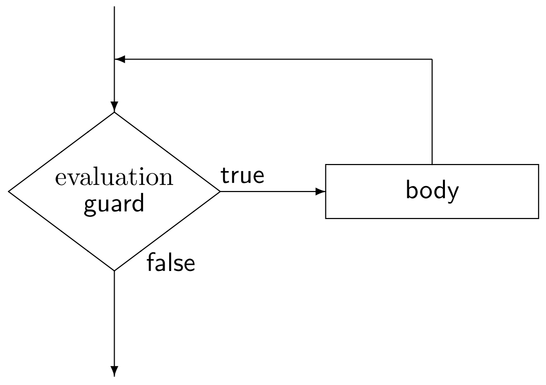


Figure 2.1: A schematic representation of the **while**-statement

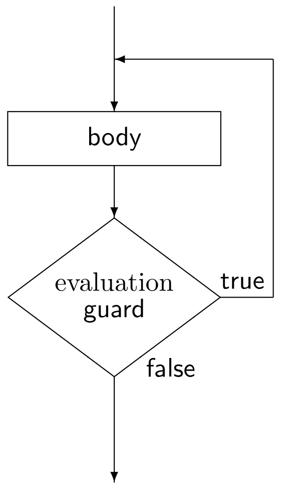


Figure 2.2: A schematic representation of the **do-while**-statement

What can we say about the value of the variables from the guard at the beginning of the body of the loop (the case in which the guard holds), and the state immediately after the while? After execution of the loop, it is certainly true that the guard is false, otherwise the loop would not have stopped. Conversely, at the beginning of the body we know that the guard is true, otherwise the loop was never entered. This reasoning is shown in the following code fragment:

```
while (guard) {
    /* At this point, the guard holds (is true) */
    body;
}
/* At this point, the guard does not hold (is false) */
```

A concrete example is:

```
i = 1;
while (i != 11) {
    /* At this point: i != 11 */
    printf("%d*2=%d\n", i, 2*i);
    i++;
}
/* At this point: i==11 */
```

2.1.2 The **do while**-statement

The **do while**-statement is used if we want the body of the loop to be executed at least once (regardless of the initial value of the guard). Its syntax is

```
do {
    body;
} while (guard);
```

After the body is executed, the boolean expression guard is evaluated. If the guard is true, the loop will be executed again. Otherwise, the loop terminates. This loop construct is depicted schematically in Figure 2.2.

Note that we cannot be sure that the guard holds at the beginning of the body, since it might not be true on the first iteration. The only thing that we can say with certainty is that the guard is false after execution of the loop:

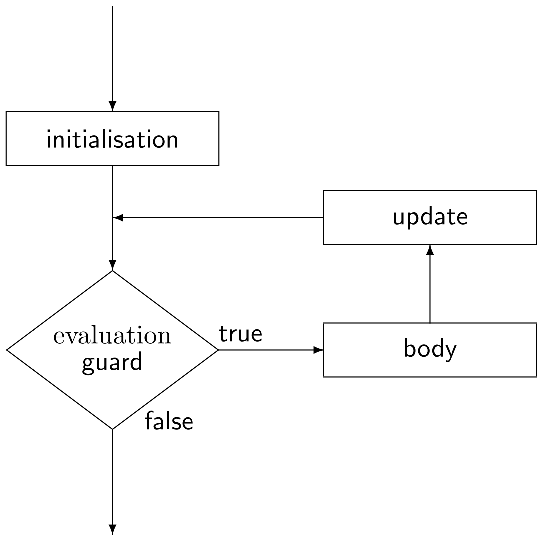


Figure 2.3: A schematic representation of the **for**-statement

```
do {
    /* At this point the guard may be false only in the first iteration. */
    body;
} while (guard);
/* At this point, the guard does not hold (is false) */
```

A concrete example is:

```
i = 1;
do {
    printf("%d*2=%d\n", i, 2*i);
    i++;
} while (i != 11);
/* At this point: i==11 */
```

2.1.3 The for-statement

The most general form of the **for**-statement is:

```
for (initialisation; guard; update) {
    body;
}
```

Here, initialisation and update are statements. The semantics of the **for**-loop is explained easiest by first converting it into an equivalent **while**-loop. The general form of the **for**-loop is equivalent to the following **while**-loop:

```
initialisation;
while (guard) {
    body;
    update;
}
```

From this equivalent code, it is easy to see what the semantics of a **for**-loop is. First, the initialisation is executed. Next, the body is executed as long as the guard evaluates to true. In each iteration, directly after execution of the body the update is executed. Usually, the update modifies one or more variables that eventually make the guard false. The **for**-loop construct is depicted schematically in Figure 2.3.
A typical use of the **for**-loop is iterating over a fixed set of integers, for example the half-open interval [0..10):

```
sum = 0;
for (i = 0; i < 10; i++) {
    sum += i;
}
```

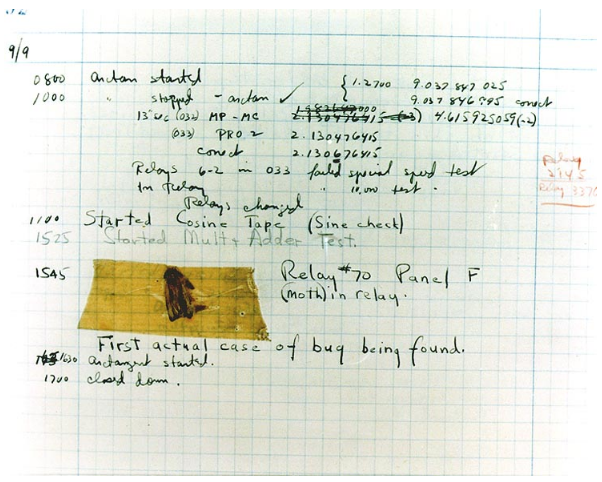
We have seen that we can rewrite any **for**-loop as a **while**-loop. However, the converse is also true. It is possible to convert a **while**-loop into an equivalent **for**-loop as well because the initialisation, guard and update of a **for**-loop are optional and can thus be omitted. So, every **while**-loop can be converted easily in a **for**-loop as follows:

```
for ( ; guard; ) {
    body;
}
```

Thus, in a way we could say that the **while**-loop and the **for**-loop are equivalent. The difference between the two loops is purely conceptual.

The choice of which type of loop to use is not an exact science, but merely a matter of preference. A **for**-loop is typically used to iterate over a fixed set of values. On the other hand, the **while**-loop is typically used to execute the body as long as the guard holds.

If you write a **for**-loop with an omitted initialisation, you can probably better use a **while**-loop. If you write a **while**-loop that needs to initialise a variable first, that is modified at the end of the body, then you can probably better use a **for**-loop.


9/9

0800 antan started

1000 " stopped - antan ✓ {1.2700 9.037 847 025
9.037 846 995 const

13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415 (-3)~~ 4.615925059(-2)

(033) PRO 2 2.130476415

const 2.130676415

Relays 6-2 in 033 failed special speed test
in relay " 11.000 test .

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Mult + Adder Test.

1545 Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.

~~143~~ 1630 antangent started.

1700 closed down .

Relay 2145
Relay 3370

Figure 2.4: Bug found. From the logbook of the Mark II Aiken Relay Calculator at Harvard University, 9 September 1945.

2.2 Debugging a program

The process of detecting and correcting errors in computer programs is called *debugging*. It is common to call a programming error a *bug*. This term stems from the time when computers were very big machines that were equipped with electrical relays that were not encapsulated in a housing. This allowed small bugs such as flies or moths to get caught between the contacts of the relay. Naturally, this led to failures. Originally, debugging was the process of finding and removing bugs that caused machine malfunctioning (Figure 2.4).

Finding an error in a program can be difficult and time-consuming. In simple situations, the programmer can print the values of variables by inserting printf statements in the program. By inspecting the output of these auxiliary printf statements, the programmer may be able to track down the error and correct it. For example, the extra printf statement in the following program fragment helps to find the 'division by zero'-error since the program crashes after it prints i=0.

```
cnt = 0;
for (i=n ; i >=0 ; i−−) {
  printf("i=%d\n", i); /* debug statement */
  if (n%i == 0) {
    cnt++;
  }
}
```

This debugging method is typically used when there are only few variables that need to be displayed on the screen in order to find the error. When there are many variables involved, this method can become confusing due to the large amount of produced output. Especially in combination with loops, this method generates too much (debug) output to be useful.

2.2.1 Conditional compilation

A simple way to determine the impact of a piece of code is to use a technique that is called *conditional compilation*. In C, it is possible to compile fragments of the source code conditionally (or omit them conditionally). Often, conditional compilation is used to compile code fragments that are developed for a specific platform (computer type) or operating system, or to compile debug statements. Conditional compilation can be used as follows:

```
  cnt = 0;
  for (i=n ; i >=0 ; i--) {
#if 1 /* 1=code gets compiled, 0=code is omitted */
    printf("i=%d\n", i); /* debug statement */
#endif
    if (n%i == 0) {
      cnt++;
    }
  }
```

When we compile the above code, the line with the printf statement will be included in the program. However, if we replace #if 1 by #if 0, then the line with the printf statement is omitted. So, this is a special way of commenting-out regions of code. By systematically turning on and off fragments of code, we can try to locate bugs. If you want to know more about conditional compilation, you are recommended to read the section A. 12 on the C preprocessor (cpp) from the book "The C Programming Language" by Kernighan and Ritchie

2.2.2 Programming with assertions

When creating a program, we often use some boolean predicate P that at a particular point in the program must be true. We often write P then as a comment at that location in the program. With the help of assert it is possible to check during execution of the code if the predicate P really holds. In order to use assert, you need to include the header file assert.h at the top of your program.

Here is a simple example of the use of assert:

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

int main(){
  float sum = 0;
  int n = 10;
  while (n != 0) {
    n--;
    assert(n!=0);
    sum += 1.0/n;
  };
  printf("sum(10)=%f.\n", sum);
  return 0;
}
```

The program is intended to compute: $\frac{1}{10} + \frac{1}{9} + \frac{1}{8} + \frac{1}{7} + \frac{1}{6} + \frac{1}{5} + \frac{1}{4} + \frac{1}{3} + \frac{1}{2} + \frac{1}{1}$. However, if you save the above code in a file demo.c and compile and run it, you will get an error message (which is similar to):

```
a.out: demo.c:9: Assertion `n!=0' failed.
Aborted
```

Apparently, the error is a division-by-zero bug. The obvious solution is to move the statement n-- to the end of the loop.

The use of assert has a clear advantage over the use of debug printf statements: it does not produce an enormous amount of useless output, and stops directly when an error is detected.

Once you are done debugging your program using assertions, there is no need to remove all lines that contain an assertion to get the final version of your program. It suffices to add the line

```
#define NDEBUG
```

before the line that includes the file `assert.h`.

Tip!

The following idiom is handy, since you can easily turn debgugging with assertions on and off:

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

#if 1 /* 0 for assertions on, 1 for assertions off */
  #define NDEBUG
#endif
#include <assert.h>

/* here your code */
```

2.3 Tutorial exercises

2.3.1 Numerical types

Assume the following declarations:

```
short a = 350;
int b = 3;
int c = 10;
long d = 1;
float x = 391;
double y = 0.71;
double z = 0.35;
```

Determine for each of the following assignments the type of the expression on the right-hand side of the =-sign. Also specify whether the assignment of an expression of this type to the variable on the left-hand side is 'safe', i.e. whether the type of the variable on the left-hand side is at least as large as the type of the expression. If the assignment is safe, determine also the value of the right-hand side of the assignment.

1. `d = a * b;`
2. `c = a / b;`
3. `y = c / b;`
4. `y = b + x;`
5. `a = a + 1;`
6. `a++;`
7. `b = a + 1;`
8. `d = 100 * (x - y);`
9. `x = (float)(a / b);`
10. `z = (float)a / c;`
11. `a = (int)y + x;`
12. `a = (int)(y + x);`
13. `a = (short)(y + x);`
14. `c = (int)a / z;`
15. `c = a / (int)z;`
16. `c = (int)(a / z);`

2.3.2 Divisors

Write a program that asks the user to type in a positive integer and then prints all positive integer divisors of that number.

2.3.3 Exponentiation

Write a program fragment that computes b^e given an integer b and a positive integer e. Of course, you are not allowed to use built-in functions from `math.h`.

2.3.4 Integer logarithm

The *integer logarithm* of an integer number n (with $n \geq 1$) to an integer base b (with $b > 1$) is the number of times b can be multiplied by itself without exceeding n. In other words, it is the largest integer m such that $b^m \leq n$.

Write a program that asks the user to type in two integers n and b (with $n \geq 1$ and $b > 1$) and then prints the integer logarihm of n to the base b. Of course, you are not allowed to use built-in functions (such as log) from `math.h`.

2.3.5 The inverse of the factorial function

The *factorial* of a non-negative integer number n (notation $n!$) is defined as the product

$$n! := \prod_{k=1}^{n} k$$

Note that $0! = 1$, since the product of an empty domain is 1 by definition.

The inverse of the factorial function is defined as the function caf:

caf(x) = largest integer n such that n! <= x.

Example:

```
caf(1) == 1
caf(24) == 4
caf(200) == 5
```

Write a program that, given a non-negative integer number n on the input, prints caf(n) on the output.

Chapter 3

Procedural programming

3.1 Functions

A *function* is a program fragment that performs some logically coherent task that can be used from different locations in the program.

The description of a function, just like variables, is called a *function declaration*. We can use a function by 'calling' it: this is called a *function call*. We can supply a function with values via *parameters*: the actual supplied values are called *arguments*. It is common that a function returns a *function value* (e.g. an **int** value), however some functions do not return anything.

3.1.1 Function declaration

The declaration of a function consists of a few parts. In general, a function declaration has the following form:

```
type functionName(type1 parameter1, ..., typeN parameterN) {
    bodyStatements;
}
```

First we specify the type of a function. The *type* of a function is the type of the *result* or *return value* of the function. The type of a function is often referred to as its *return type*. If the function does not return a result, the type is specified as **void**. This type of function is called a **void** *function* or a *subroutine*.

The type of the function is followed by the name of the function. The naming rules for functions are the same as for variables, plus the following extra conventions. A (void) function without result contains in its name a verb in the imperative form. For example, printSequence would be a good name for a function that prints a sequence of numbers. The name of a function that returns a result usually gives a description of the result obtained. For example, isPrime would be a good name for a boolean function with one argument, that returns 1 (true) if its argument is a prime number, otherwise 0 (false).

The list of parameters is made up of a sequence of variable declarations separated by commas. The list is enclosed by parentheses. If the function has no parameters, this list is empty (i.e. only a pair of parentheses).

The parameter list is followed by the *body* of the function: the actual code. In the body, we can use the parameters like ordinary variables. These variables are *local variables* within the function: changing the value of a local variable does not affect the value of the argument that was supplied when the function was called. This mechanism is known as *'call-by-value'*.

Void functions

A function that does not return a result has the return type **void**. For example, the following function explains to the user that he/she should type in an integer from the interval [lwb,upb).

```
void printIntervalMessage(int lwb, int upb) {
    printf("Please type a number from the interval [%d,%d).\n", lwb, upb);
}
```

As you can see, the parameters lwb and upb are used as if they were ordinary variables. The function does not return a result. Hence, the return type is **void**.

Functions that return a result

Within the body of a function that returns a result, we use the statement **return** to stop execution of the function body immediately and return the function result. A **return** statement has the following form:

```
return expression;
```

where expression must be an expression that has the same type as the return type of the function. Within the body of a function that does not return a result (a **void** function) we can also use **return** to terminate execution of the body, but of course without returning a value:

```
return;
```

In a **void** function it is not necessary to use a **return** statement, since it is implicitly the last statement of such a function. So, in a **void** function the statement **return** is usually only used as an early termination in a loop.

A simple extension of the function printIntervalMessage would be to read in a number and return it. Of course, this would mean that the function is no longer a **void** function, since it should return an **int**. So, we arrive at the following function numberFromInterval:

```
int numberFromInterval(int lwb, int upb) {
    int number;
    printf("Please type a number from the interval [%d,%d).\n", lwb, upb);
    do {
        scanf("%d", &number);
        if (number < lwb) {
            printf("%d < %d: try again\n", number, lwb);
        } else if (number >= upb) {
            printf("%d >= %d: try again\n", number, upb);
        }
    } while ((number < lwb) || (number >= upb));
    return number;
}
```

You can see that the return type of the function is an **int**: an integer is produced as a result. The loop in the body of the function is executed until a number is entered within the required limits. When the loop terminates, the last entered value is returned as the result of the function.

It is allowed to use the statement **return** within a loop. In that case, the loop (and the function) will stop immediately, and a value is returned. Whether you want to use a **return** statement within the body of a loop is a matter of taste. Some programmers consider it bad style, while others use this feature frequently. It is up to you to decide. An alternative implementation of numberFromInterval using this feature would be:

```
int numberFromInterval(int lwb, int upb) {
    int number;
    printf("Please type a number from the interval [%d,%d).\n", lwb, upb);
    while (1) {
        scanf("%d", &number);
        if ((lwb <= number) && (number < upb)) {
            return number;
        }
        printf("%d is not in the interval [%d,%d): try again\n", number, lwb, upb);
    }
}
```

3.1.2 Function call

Functions can be *called* by writing the function name and a (possibly empty) list of *arguments*. The value of the arguments are copied into the parameters of the function. Since the values are copied, the original values cannot change by a function call. Clearly, the number of arguments must be equal to the number of parameters of the function. Each argument must be an expression of a type that is compatible with the type of the corresponding parameter in the declaration of the function.

The call to a **void** function is an ordinary statement, while calling a function with a result is an expression of the same type as the function type which can thus be assigned to a variable. For example, the following **void** function

```
void printInt(int n) {
    printf("n=%d\n", n);
}
```

can be called as follows:

```
printInt(42);
```

The function square

```
float square(float x) {
    return x*x;
}
```

can be used to compute $ax^2 + bx + c$ as follows:

```
y = a*square(x) + b*x + c;
```

Note that you are already used to calling functions: printf, scanf, and sin are examples of built-in functions that we have used before.

3.1.3 Scope

The *scope* of a variable is the region in the code in which the variable 'exists'. We distinguish two types of variables: *global variables* and *local variables*. A global variable is declared outside any function, usually at the top of the source file (often directly after the include files). The scope of a global variable is the entire program. This is different for local variables: in general we can say that the scope of a local variable is the region of code from the declaration of the variable up to the end of the closing brace of the code block in which the variable is declared. A local variable may have the same name as a variable that is already present in the surrounding scope. In that case, the variable from the surrounding scope is temporarily invisible. This phenomenon is called *shadowing*. If the execution of the program reaches the end of the scope of the local variable, then the original variable becomes visible again.
Consider the following example and try to figure out what the output of this program will be (do not look directly at the answer):

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

int x = 1;

void function1() {
  printf("%d\n", x);
  x++;
  printf("%d\n", x);
}

void function2() {
  int x = 3;
  printf("%d\n", x);
  x++;
  printf("%d\n", x);
  if (x == 4) {
    int x = 5;
    printf("%d\n", x);
    x++;
    printf("%d\n", x);
  }
  printf("%d\n", x);
}

int main(int argc, char*argv[]) {
  printf("%d\n", x);
  function1();
  printf("%d\n", x);
  function2();
  printf("%d\n", x);
  return 0;
}
```

As you can see, the variable x is declared three times. This is no problem, since the scopes are different. However, shadowing will occur. The first delaration at the top of the program is outside any function and is therefore a global declaration. This global variable x is therefore visible everywhere in the program where it is not shadowed by a local variable with the same name. We say that the scope of x covers the entire program.

In the main function the value of the global variable x is printed, so the output will be 1, as you would expect. Then the function function1 is invoked. Since this function does not have any local variable in its scope, a reference to x within this function must be a reference to the global variable x. The function prints first the value of x (i.e. 1), then it increments x and prints its value again (i.e. 2).

After execution of function1, the execution of the program returns to the main function. This function prints x as well. Since function1 incremented the value of x, the value 2 is printed.

Then the function function2 is invoked. In this function another variable x is declared. This variable is local, and has a scope that ends at the last closing brace of the function function2. The local variable x shadows the global variable x. The local variable x has initially the value 3, and so 3 is printed. Next, the local x is incremented and thus 4 is printed. Since x==4 evaluates to true, the body of the **if**-statement is executed. Here we see that it is also possible to introduce local variables in a local code block. So, again a new local variable x is declared that shadows the local variable x from the surrounding scope. Of course, the body of the **if**-statement prints 5, followed by 6.

Now it gets interesting. We leave the code block of the **if**-statement. The local variable x which is declared in the body of the **if**-statement gets *out of scope*, so the variable disappears (ceases to exist). As a result, the local variable x from the surrounding scope becomes visible again. This variable still has the value 4, so the value 4 is printed. Then the function function2 ends and the local variable x disappears as well. We are left with the only still visible global variable x, which still has the value 2. Thus, finally 2 is printed in the function main.

In conclusion, the program produces the following output:

```
1
1
2
2
3
4
5
6
4
2
```

3.1.4 Parameter passing mechanism: call-by-value

The parameters of a function can be regarded as local variables: they exist only inside the function. We make a distinction between *formal parameters* and *actual parameters*. Formal parameters are the parameters that occur in the function definition. Actual parameters (also known as arguments) are values (expressions) that are passed by the caller. For example, in the following function n is a formal parameter.

```
void printNextInt(int n) {
    n++;
    printf("%d\n", n);
}
```

When we call this function, the argument that we pass to this function is the actual parameter (which is a value). For example:

```
printNextInt(41);
```

Of course, we could also write:

```
a = 41;
printNextInt(a);
```

In both cases, the *value* 41 (which is not a variable!) is the actual parameter of the function printNextInt. You can think of this as follows: the value 41 is *copied* in the local variable n (the formal parameter) of the function printNextInt before the code of the function is started. This principle is called *call-by-value*. In C the passing of parameters, the so-called *parameter passing mechanism*, always follows this call-by-value principle.

An important consequence of this is the following. When a parameter is passed by value, then value of the actual parameter is copied into the formal parameter. So, when the function modifies the value of the formal parameter, it modifies a copy and not the original actual parameter. Hence, a function cannot change the value of the actual parameter via the parameter passing mechanism. So, the program fragment

```
a = 41;
printf("%d\n", a);
printNextInt(a);
printf("%d\n", a);
```

will produce the following output:

```
41
42
41
```

The increment of the formal parameter n has no effect on the variable a that was used as the actual parameter.

Suppose we want to make a function swap with two parameters such that the call swap(a, b) interchanges the values of the variables a and b. If you understand the principle of call-by-value then you know why the following function does not work:

```
void swap(int a, int b) {
  int h = a;
  a = b;
  b = h;
}
```

This function is perfectly valid C, and will be accepted by the compiler without any complaints. However, since modification of the formal parameters has no effect on the actual parameters, this function does basically 'nothing'. So, the question is how to implement the desired functionality? To answer this question, we first need to get acquainted with a special type of variables: pointers.

3.1.5 Pointers

The memory of a computer can be imagined as a big table, with only two columns. The first column contains addresses (or memory locations) while the second column contains values (bytes, i.e. groups of 8 bits) that are stored in the corresponding memory locations. A 1GB memory (one gigabyte) has approximately one billion memory locations (addresses), each containing one byte of data.

If you declare a variable, then the compiler reserves an address in the memory of the computer. This address is then used to store values that are assigned to the variable. Thus, with the declaration

```
int i;
```

we reserve 32 bits (on a PC) of memory for the variable i. The compiler chooses the memory location, for example address 10000. Since an **int** has 32 bits, the compiler not only reserves the memory location (byte) at address 10000, but also the bytes at memory locations 10001, 10002 and 10003.

In C, it is possible to get the address of a variable by prefixing the variable name with an ampersand (the &-character). The ampersand is called the *address operator*. For example, we can print the address of the variable i as follows:

```
printf("The address of variable i is %p.\n", &i);
```

Note that we use the format specifier %p of printf. It allows you to print an address. Normally, it is irrelevant to know the exact location (address) of a variable, so %p is rarely used.

It is not possible, without casting, to assign the address of a variable to an ordinary **int**, but it is possible to assign an address to a special kind of variable, a *pointer variable*. A pointer variable contains an address and 'points' to the data that is stored at that address. Hence the name *pointer*.

In C we declare a pointer variable by prefixing its name by an asterisk (the *-character). For example, to declare a pointer variable ptr that points to an **int**, we would use

```
int *ptr;
```

The asterisk tells the compiler that we want to use a pointer variable. The type **int** tells the compiler that we want to interpret the data that is stored at address ptr as an **int**: the pointer points to an integer.

Note that a *global* variable that has not been initialised automatically gets the initial value zero. An uninitialised *local* variable contains some arbitrary value. Reading an uninitialised local variable prior to assigning a value to it results in undefined behaviour. This is also true for *pointer* variables. A global uninitialised variable is initially set to address 0. To make a clear distinction with the normal **int** value 0, a special symbolic name is introduced for the zero-pointer: the NULL pointer. The value NULL is usually used to denote that a pointer does not point to anything (since address zero is meaningless, it is very likely reserved by the operating system). You can use a simple assignment to set a pointer to NULL:

```
ptr = NULL;
```

The value NULL is a valid address, just like any other. So, you can test whether a pointer is NULL or not.

```
if (ptr == NULL) {
  printf("The pointer has value NULL\n");
} else {
  printf("The pointer points to some memory location.\n");
}
```

Now, assume that we want to store the address of the **int** variable i in the pointer ptr. We can use the address operator & to accomplish this:

```
ptr = &i;
```

Now, ptr points to i.

Besides the address operator, there exists the indirection operator which is denoted by an asterisk (the *-character). Do not get confused by the double role of the asterisk. From the context it is clear whether it is used as the multiplication or indirection operator. The indirection operator * yields the value stored at the location that a pointer points to. For example, we can print the value that ptr points to as follows:

```
printf("The value stored at address %p is %d.\n", ptr, *ptr);
```

Since ptr points to the value of i, we can print i as follows in a valid, but clumsy, way:

```
printf("i=%d\n", *ptr);
```

The indirection operator can also be used on the left-hand side of an assignment:

```
*ptr = 7;
```

This assignment is interpreted as follows. The value 7 is stored in the memory location that ptr points to. This assignment has an effect that you may not have guessed. Since ptr points to the address where the variable i is stored, the assignment *ptr = 7 changes the value of i! In other words, after this assignment, the variable i has the value 7. So, the indirection operator * on the left-hand side of an assignment changes the value stored at the location that the pointer points to, and not the value of the pointer itself! In fact, the assignment above is equivalent with the much simpler assignment:

```
i = 7;
```

It is allowed to assign a specific address to a pointer, for example:

```
ptr = 1000;
```

This assignment stores the address 1000 in the pointer ptr. This is permitted, but is rarely useful. After all, we do not know what is stored at address 1000, so making a pointer to this address is probably meaningless. Such an assignment makes only sense if we know that a specific memory address is used for communication with peripheral devices. For example, a specific memory location is the start address of the pixels on the screen (the so-called frame buffer). Knowing this address enables a system programmer to write device drivers that communicate with peripherals directly. This goes far beyond the scope of this introductory course in C, so for the time being we will not assign concrete addresses to pointers. However, it is useful to know that the gcc compiler will respond with a warning if you try to compile the above pointer assignment. You might accidentally produce such an assignment yourself.

```
warning: assignment makes pointer from integer without a cast
```

In a way, the warning makes sense: on the left-hand side of the =-sign there is an address whereas the right-hand side is a number. Although adresses are numbers, it is better to explicitly cast the **int** value to a pointer of type **int**. The following equivalent assignment is accepted by the compiler without a warning.

```
ptr = (int *)1000;
```

To conclude a small test. Try to predict what will be the output of the following program:

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

int main(int argc, char *argv[]) {
  int a = 1;
  int b = 2;
  int *pa;
  int *pb;
  int h;
  printf ("a=%d, b=%d\n", a, b);
  pa = &a;
  pb = &b;
  h = *pa;
  *pa = *pb;
  *pb = h;
  printf ("a=%d, b=%d\n", a, b);
  return 0;
}
```

3.1.6 Function with formal pointer parameters: call-by-reference

The parameter mechanism of C uses the call-by-value principle. This prevented us from implementing the function swap straight-forwardly. Sometimes, we really want to modify the value of one or more arguments to a function. Many programming languages (e.g. Pascal and C++) have a second parameter mechanism which is called *call-by-reference*. This mechanism does not copy the value of the arguments but passes through references to the variables, so that the function modifies the contents of the actual arguments (and not copies).

Although the parameter mechanism of C is strictly call-by-value, the call-by-reference mechanism is effortlessly simulated with the aid of pointers. We can use pointers as formal parameters. The pointers are values that cannot be changed by the function, but the data that the pointers point to can be changed through the indirection operator. In fact, we use the same 'trick' that we used to modify the variable i by assigning a value to *ptr (see assignment *ptr=7).

So, we can implement the function swap using pointers as formal paramters.

```
void swap(int *pa, int *pb) {
  int h = *pa;
  *pa = *pb;
  *pb = h;
}
```

Clearly, we must call this function with pointer arguments:

```
swap(&a, &b);
```

The call swap(a,b) is wrong and will produce an error (or warning) message from the compiler.

In fact, you already know that you sometimes need to use addresses in combination with function calls. When we use the function scanf to read input from the keyboard, it is necessary to prefix variables with an ampersand. Until know, it was completely unclear why. Now you know.

Maybe you wonder whether it is possible to make a function that swaps two pointers instead of two **int**s. This is indeed the case. You simply apply the same principle: we introduce pointers to pointers by using two asterisks. The following function swaps two **int**-pointers:

```
void swapPointers(int **ppa, int **ppb) {
  int *h = *ppa;
  *ppa = *ppb;
  *ppb = h;
}
```

3.2 Composite data types

3.2.1 structs

In C, a number of variables can be grouped together in a composite data type. Such a composite data type is called a *struct* (from structure).

Suppose we want to do arithmetic with rational numbers (fractions) instead of **float**s. The data type rational does not exist in C, but we can build it ourselves. We represent the numerator and denominator of a fraction with two **int**s. It is more natural to think in terms of a data type fraction rather than a pair of **int**s that represent a fraction. Therefore we group the numerator and denominator in a **struct**. We can define the **struct** rational as follows:

```
struct rational {
  int numerator;
  int denominator;
};
```

Once the struct is defined, we can declare a variable of this type as follows:

```
struct rational r;
```

The variable r consists of two **int**s. We can address the elements of the **struct**, the so-called *fields*, using the notation r.numerator and r.denominator.
For example, the fraction $\frac{1}{2}$ can be constructed as follows:

```
r.numerator = 1;
r.denominator = 2;
```

It is possible to initialise a **struct**-variable directly with its declaration. An alternative way to construct the fraction $\frac{1}{2}$ is:

```
struct rational r = {1,2};
```

Once a **struct**-variable has been initialised, you can assign it as a whole to another variable of the same type. For example, the following code asigns to the variables a and b the fraction $\frac{1}{2}$:

```
struct rational a = {1,2};
struct rational b;
b = a;
```

So, the above assignment b=a is a convenient (and preferred) way of writing:

```
b.numerator = a.numerator;
b.denominator = a.denominator;
```

3.2.2 typedefs

Often, it can be handy to introduce a **typedef**, which is basically an alias for an already existing type. For example, if you prefer to write integer instead of **int**, then you can achieve this by defining the alias integer as follows:

```
typedef int integer;
```

Now, you can declare a variable of the type integer. Of course, the type **int** still exists, so you can actually use both.

```
integer x;
int y;
```

The variables x and y are completely compatible. A function that has a formal parameter of type **int** can be called with the actual parameter x as well as y.

Of course, you can also define a type-alias for **struct**s. We can introduce the type rationalNumber as an alias for **struct** rational:

```
typedef struct rational rationalNumber;
```

Now, you can declare a variable of the type rationalNumber:

```
rationalNumber r;
```

Tip!

If you define a **typedef** for a **struct**, then it is a bit clumsy that the **struct** has a name that we probably want to use for the **typedef**. This is not a problem, since C accepts that **struct**s and **typedef**s have the same name. So, we could have defined the **typedef** rational as follows:

```
typedef struct rational rational;
```

In fact, we can even combine the definition of the **struct** and the new type-alias. In the case at hand, this would look like:

```
typedef struct rational {
  int numerator;
  int denominator;
} rational;
```

Sometimes, we do not even bother to give the **struct** a name. In that case, we can write:

```
typedef struct {
  int numerator;
  int denominator;
} rational;
```

3.2.3 Pointers and structures/typedefs

As you might have guessed, it is possible to declare a pointer that points to a **struct** or **typedef**.

```
struct rational *rptr;
rationalNumber *bptr;
```

However, there is something special about the combination of pointers and structures/typedefs. If we want to access one of the fields of the structure that the pointer points to, then we use the notation −> instead of . (dot). For example:

```
rptr->numerator = 1;
rptr->denominator = 2;
bptr->numerator = 2;
bptr->denominator = 3;
```

It is possible to continue to use the dot notation to access the fields, but then you have to make use of the indirection operator. This is quite unusual: it is not really C-stylish. However, it is valid C. So, the code fragment above is equivalent with the following code:

```
(*rptr).numerator = 1;
(*rptr).denominator = 2;
(*bptr).numerator = 2;
(*bptr).denominator = 3;
```

3.2.4 Abstract Data Types (ADTs)

It is considerd good programming style to define **struct**s/**typedef**s together with a set of functions (operations) that operate on variables of the corresponding type. Such a combination of a data type with the corresponding operations is called an *abstract data type* or *ADT*. These data types are of a higher level of abstraction than the standard builtin-types: the user of the data type is only concerned with the available operations on the data type, not with the actual representation of the data type.
For example, consider the following **typedef**:

```
typedef struct rational rational;
```

The user of the data type rational is not interested at all how rational numbers are implemented. He/She is only interested in using the data type and the corresponding operations. In other words, the actual implementation remains abstract to the user.

For the abstract data type rational, it is natural to define the following operations: addition, subtraction, multiplication, and division. We probably also want an operation that prints a rational number on the screen. These operations can be implemented as follows:

```
void add(rational *a, rational b) {
  /* implements: a = a + b */
  a->numerator = a->numerator*b.denominator + b.numerator*a->denominator;
  a->denominator = a->denominator*b.denominator;
}

void sub(rational *a, rational b) {
  /* implements: a = a - b */
  a->numerator = a->numerator*b.denominator - b.numerator*a->denominator;
  a->denominator = a->denominator*b.denominator;
}

void mult(rational *a, rational b) {
  /* implements: a = a*b */
  a->numerator = a->numerator*b.numerator;
  a->denominator = a->denominator*b.denominator;
}

void div(rational *a, rational b) {
  /* implements: a = a / b */
  a->numerator = a->numerator*b.denominator;
  a->denominator = a->denominator*b.numerator;
}

void printRational(rational a) {
  /* prints a rational number on the screen */
  printf ("%d/%d", a.numerator, a.denominator);
}
```

These ADT-operations can be used to compute the expression $\frac{3}{8}(\frac{1}{2}+\frac{3}{4})/\frac{1}{5}$ as follows:

```
rational a = {3, 8};
rational b = {1, 2};
rational c = {3, 4};
rational d = {1, 5};

add(&b, c);
mult(&a, b);
div(&a, d);
printRational(a);
```

Note that in the above code snippet, we do not make use of knowledge about the internal representation of fractions. This is the whole point of an abstract data type. If at some point in time you decide to represent fractions by some other data structure, it is sufficient to only re-implement the operations of the ADT. Any code that makes use of these ADT operations only should not change at all.

At this point of time, this may not seem so important. However, in the future you will build larger programs with thousands of lines of code (or longer). In such projects, the value of ADTs becomes clear very soon.

3.3 Tutorial exercises

3.3.1 Square tester

Write a function with one integer parameter that tests whether the value of the parameter is the square of an integer or not. Of course, you are not allowed to use the function `sqrt` from the `math.h` include-file.

3.3.2 Majority vote

The expression `(a || b || c)` evaluates to a non-zero value (true) if at least one of the expressions `a`, `b` or `c` is non-zero.
Write a function with three parameters of type **int** that returns a non-zero value (true) if and only if at least two of its parameters are non-zero.

3.3.3 Multiples

Write a function `isMultipleOf` so that the call `isMultipleOf(x, y)` returns true if and only if `x` is an integral multiple of `y`. The parameters `x` and `y` are of type **int**.

3.3.4 Leap year

A year is a *leap year* if it is divisible by 4, but not by 100. Exceptions are years that are divisible by 400 (these are leap years).
Write a function `isLeapYear` that determines whether a year is a leap year or not.

3.3.5 First digit

Write a function `firstDigit` that returns the first digit of a non-negative integer number. Examples:

```
firstDigit(648768) == 6
firstDigit(5552200) == 5
```

3.3.6 Triangle

Write a function `showTriangle` with two parameters: a character `ch` and a positive integer `n`. The function should print a triangular diagram as follows: `n` copies of `ch` on the first line, `n−1` copies of `ch` on the second line, etc.
For example, the output of `showTriangle('+', 4)` must be:

```
++++
+++
++
+
```

3.3.7 Number of solutions of a quadratic equation

Write a function

```
int numSolutions(int a, int b, int c)
```

that returns the number of unique real solutions of the quadratic equation $a \cdot x^2 + b \cdot x + c = 0$.

3.3.8 Palindrome number

A *palindrome* is a word that reads the same in forward and reverse direction. Some examples are: radar, level, rotor, kayak, madam.

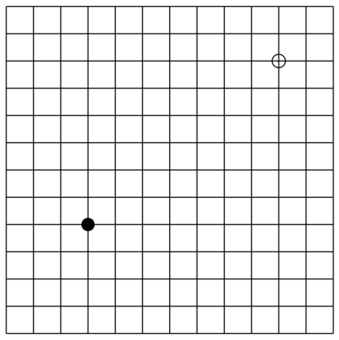
A *palindrome number* is a number that has the same property: it can be read in both directions. An example is the number 12321. Write a complete program that asks the user to type in a non-negative number and then prints whether the number is a palindrome number or not.

3.3.9 Manhattan distance

The street map of Manhattan is simple. The streets have only one of two possible directions: north to south or west to east. The distance between two streets running parallel is called a block.

This is very similar to a grid that we know from mathematics. We therefore number the crossings of the streets the same as we do in mathematics, viz. as a pair of integer coordinates (x, y).

Write a program that prompts the user to type in the coordinates of two crossings and then prints the distance between these two locations in terms of the number of blocks between them. This distance is called the *Manhattan distance* or *city-block distance*, which is clearly different from the *Euclidean distance* (straight-line distance as the crow flies).



Chapter 4

Arrays and memory management

4.1 Arrays

In an ordinary variable you can store a single value. An *array* allows you to store a sequence of values of the same type. An array is in fact a numbered list, with a variable at each position in the list. The number of a position is called the *index* of that position. The values that are stored in the array are called the *elements* of the array.

4.1.1 Declaration and initialisation of arrays

We can declare an array in which we can store one hundred **int**s as follows:

```
int list[100];
```

In this example, the name of the array is list. For the naming convention of arrays we use the same rules as those for ordinary scalar variables. After declaring the array, we have reserved enough memory space to store hundred **int**s: we say that the *array size* is 100. If the array is declared globally, then all elements of the array are initialised to zero. If the array is declared locally, then this is not the case and the array contains arbitrary data.

We can combine the declaration and the initialisation of an array in one line of code:

```
int digits[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

After this declaration, we have the availability of an array digits which can store ten **int**s. Initially, the array is filled with the digits 0 to 9.

When we use direct initialization of an array, the compiler is able to determine by itself the size of the array. Therefore, it is allowed to omit the explicit specification of the size of the array. For example, the above declaration is equivalent to the following declaration:

```
int digits[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

The compiler is able to determine that the size of the array is 10 elements.

Of course, it is not allowed to specify an explicit array size that is smaller than the number of initialisation elements. So, the following declaration is wrong:

```
int digits[9] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}; /* wrong ! */
```

However, it is allowed to specify an explicit array size that is bigger than the number of initialisation elements. If the array is declared globally, then the remaining elements will be initialised with zeroes. If the array is declared locally, then the remaining elements will not get initialised (i.e. contain arbitrary value). An example is the following declaration of the global array digits:

```
int digits[10] = {0, 1, 2, 3, 4, 5}; /* ok ! */
```

Since digits is declared globally, this declaration is equivalent to the following declaration:

```
int digits[10] = {0, 1, 2, 3, 4, 5, 0, 0, 0, 0};
```

4.1.2 Array indexing

Accessing array elements is called *indexing*. We access an array element with index i of an array a using the notation a[i]. Note that the index of the first element of an array is always 0. So, the last element of an array of size n has index n−1.

Alert!

An array containing n elements has indices from the half-open interval [0,n). If you try to access the array at an index i that is outside this interval (i.e. i<0 or i>=10), unspecified behaviour may result: sometimes the program stops (crashes) immediately, sometimes program execution continues and (unexpected) things may go wrong later on.
Indexing an array with an invalid index is called *out-of-bounds indexing*. An example of this frequently occuring error is in the following code fragment:

```
int i, a[10];
for (i = 0; i <= 10; i++) {
    a[i] = i;
}
```

Instead of testing i<=10, we should have used the test i<10 because a[10] does not exist.

4.1.3 Multidimensional arrays

Besides one-dimensional arrays (i.e. arrays that need a single index to indicate an element), we can declare *multidimensional arrays*:

```
int matrix[3][4];
```

This declaration creates a two-dimensional array: 3 rows of 4 elements (columns). If we want to access the first element of the second row, then we can index this element with matrix[1][0] (remember, indexing starts from index 0).

Two-dimensional arrays can be initialised at declaration as well:

```
int matrix[3][4] = {{0, 1, 2, 3},
                    {0, 4, 5, 6},
                    {0, 0, 6, 7}};

int id3x3[][] = {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}; /* 3 rows, 3 columns */
```

In principle, there is no limit to the number of dimensions of an array. In practice, however, you will rarely encounter arrays that have more than three dimensions.

4.1.4 Arrays and pointers

As we now know, a pointer points to a memory location where data of a certain type is stored. Using the address operator, we can make a pointer that points to an element of an array. For example:

```
int a[3] = {1, 2, 3};
int *p = &a[0];
```

The pointer p points to the first element of the array. In other words, the boolean expression *p == a[0] is true. However, we may as well write:

```
int *p = a;
```

Both declarations of the pointer `p` are completely equivalent. This is the result of the fact that in C the name of an array is treated as if it were a pointer to the first element of the array. It is a matter of taste which format is preferred: the first declaration is more explicit, the latter is more compact. A consequence of the above is that the tests `*p == a[0]` and `p == a` are equivalent.

We can do some simple arithmetic with pointers. The only two allowed operations are addition and subtraction of an integer. Consider the following example:

```
int a[3] = {1, 2, 3};
int *p = &a[0];
int *q;
printf ("*p==%d\n", *p);
q = p + 1;
printf ("*q==%d\n", *q);
q = p + 2;
printf ("*q==%d\n", *q);
```

The output of this program fragment will be:

```
*p==1
*q==2
*q==3
```

This example shows that if `p` is a pointer to an **int**, then the expression `p+1` results in a pointer that points to the **int** value immediately following the **int** where `p` points to. This means that the expression `p+1` is numerically **sizeof(int)** greater than `p`.

Alert!

The following code fragment contains a common mistake:

```
q = p + 2*sizeof(int); /* wrong! */
printf ("*q==%d\n", *q);
```

The compiler will accept this code, but it means something else than you (probably) may expect. The intention of the code is to make `q` point to `a[2]` (remember that `p` points to `a[0]`). The size of an **int** on a standard PC is four bytes. Therefore, the pointer assignment is actually `q = p + 8`, which yields a pointer to index `8` of the array `a`. So, this pointer is out of bounds: when we try to print the value of `*q`, the program will (probably) crash.

Since we can do basic arithmetic with pointers, it is possible to use pointers as if they were arrays. As an example, the following code snippet computes the sum of the elements of an array in two different, but equivalent ways.

```
int a[3] = {1, 2, 3};
int *p = &a[0];
int i, sum;
/* first method: normal array indexing */
sum = 0;
for (i=0; i<3; i++) {
  sum += a[i];
}
printf ("sum=%d\n", sum);
/* second method: pointer addressing */
sum = 0;
for (i=0; i<2; i++) {
  sum += *(p+i);
}
printf ("sum=%d\n", sum);
```

The conclusion is that instead of a[i] we may write *(a + i). Indeed, the ANSI standard in which the C programming language is defined even uses this notation for the definition of the indexing of an array. A consequence of this is that the second method of the above code fragment can be replaced by a notational trick in which we regard pointers as arrays that can be indexed:

```
sum = 0;
for (i=0; i<3; i++) {
  sum += p[i];
}
printf ("sum=%d\n", sum);
```

A weird consequence of the equivalence of a[i] and *(a + i) is that you can even write (though not advisable):

```
sum = 0;
for (i=0; i<3; i++) {
  sum += i[p]; /* ugly, but valid C! */
}
printf ("sum=%d\n", sum);
```

4.1.5 Equivalence between pointers and arrays

From the above explanation, one might conclude that pointers and arrays are basically the same thing. This is almost true, but not completely. There are a number of similarities between arrays and pointers, but there are also differences.

The most important difference between pointers and arrays is that you cannot assign (the contents of) an array to (the contents of) another array, while you can assign a pointer to another pointer. For example, the following code is illegal.

```
    int a[10], b[10];
    a = b;    /* Wrong! */
```

As we have seen before, you can assign two pointer variables:

```
    int *p, *q;
    p = &a[0];
    q = p;
```

Pointer assignment is straightforward; the pointer on the left is simply made to point wherever the pointer on the right points to. No data is being copied: there is still just one copy, in the same place. Pointer assignment simply makes two pointers point to that one place. We call this phenomenon *aliasing*: one pointer is an alias for the other.

The similarities between arrays and pointers end up being quite useful, leading to what is called '*the equivalence of arrays and pointers*' in C. When we speak of this 'equivalence' we do not mean that arrays and pointers are the same thing, but rather that they can be used in related ways, and that certain operations may be used between them.

The first such operation is the assignment of an array to a pointer:

```
    int a[10];
    int *p;
    p = a;
```

What can the last assignment mean? In C, the result of this assignment is that the pointer p receives a pointer to the first element of a. In other words, it is as if you had written

```
    p = &a[0];
```

The second facet of the equivalence is that you can use the 'array subscripting' notation on pointers. You know that the expressions p[i] and *(p + i) are the same (provided that i is an **int**).

The third facet of the equivalence (which is actually a more general version of the first one) is that whenever you mention the name of an array a in a context where the 'value' of the array would be needed, C automatically generates a pointer to the first element of the array, as if you had written &a[0]. When you write something like

```
    int a[10];
    int *p;
    p = a + 3;
```

it is as if you had written

```
    p = &a[0] + 3;
```

which is the same as

```
    p = &a[3];
```

4.1.6 Using arrays as parameters of a function

Arrays can also be used as a parameter of a function. However, there is something special going on. Consider the following program:

```
#include <stdio.h>

void increment(int a) {
    a++;
}

void incrementFirst(int a[]) {
    a[0]++;
}

int main(int argc, char *argv[]) {
    int x = 1; /* variable of type int */
    int y[] = {1}; /* array of type int (only one element) */
    increment(x);
    printf ("x=%d\n", x);
    increment(y[0]);
    printf ("y[0]=%d\n", y[0]);
    incrementFirst(y);
    printf ("y[0]=%d\n", y[0]);
    return 0;
}
```

The output of this program will be:

```
x=1
y[0]=1
y[0]=2
```

The output is perhaps a bit different from what you expected. The first two output lines show that the function increment does not modify the contents of the actual parameters x and y[0]. We expect this since function calls use the call-by-value parameter passing mechanism. However, the function incrementFirst does change the content the of y[0] when y is used as an argument. So, when the function call to incrementFirst returns, we have y[0]==2. The cause of this phenomenon is obviously that the array itself is not passed as an argument to the function, but a pointer to the array is passed instead.

Due to the equivalence of arrays and pointers in C, the following two prototypes are the same:

```c
void incrementFirst(int a[]);

void incrementFirst(int *a);
```

Moreover, the following function calls are completely equivalent:

```c
incrementFirst(y);

incrementFirst(&y[0]);
```

4.1.7 Dynamic memory allocation

Assume that we want to make a small program that reads in a series of integers that ends with the value zero. The task is to print the series of numbers in reverse order on the output. A simple implementation would be:

```c
#include <stdio.h>

int inputSequence(int seq[]) {
  int len = 0;
  do {
    scanf("%d", &seq[len]);
    len++;
  } while (seq[len-1] != 0);
  return len-1;
}

void printSequence(int len, int seq[]) {
  while (len-1 >= 0) {
    len--;
    printf ("%d ", seq[len]);
  }
  printf ("\n");
}

int main(int argc, char *argv[]) {
  int numbers[1000];
  int length = inputSequence(numbers);
  printSequence(length, numbers);
  return 0;
}
```

However, this program has a major drawback. In the implementation we assume that the series will not exceed 1000 numbers. So, if we are confronted with a tireless user that types in more than 1000 numbers, the function inputSequence will produce an out-of-bounds error. Conversely, if the user types only 10 numbers then we have reserved much more memory than we actually need, which is wasteful.

We can avoid this by using *dynamic memory allocation*. With dynamic memory allocation it is possible to allocate at runtime the exact amount of memory that we need.

In C, the function malloc from the standard library is used to allocate a block of memory of a given length. The length of this block is, however, in bytes. The function returns a pointer to the first byte of the block of allocated memory. We already know that pointers can be indexed (just like arrays). So, using malloc it is effectively possible to create at runtime arrays of a size that is given by some expression (in program variables).

For example, let n be an integer program variable that contains a non-negative number. We can dynamically create an **int** array of size n by allocating a block of memory of n∗**sizeof**(**int**) bytes. The following function dynamicIntArray can be used to create a dynamic array of size sz:

```
int *dynamicIntArray(int sz) {
  int *ptr = malloc(sz*sizeof(int));
  return ptr;
}
```

Using this function it is easy to create a (dynamic) array of size n elements:

```
int *numbers = dynamicIntArray(n);
```

Alert!

The function malloc is from the standard C-library. It is defined in the file `stdlib.h`, so it is necessary to include `stdlib.h` at the top of your program:

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

Alert!

Next to the function malloc there exists the function calloc, which is also defined in the file `stdlib.h`. Like malloc, the function calloc is used to dynamically allocate memory. There are two diferences. The first one is mainly cosmetic: calloc has two arguments, the fist is the number of items that we want to allocate, and the second is the size of each item in bytes. So, instead of calling ptr = malloc(sz∗**sizeof**(**int**)) we use ptr = calloc(sz, **sizeof**(**int**)). The more important difference is that malloc returns a pointer to an uninitialised block of memory, while calloc returns a pointer to a block of memory of which all bytes are set to zero.

Of course it is not possible to allocate an unlimited amount of memory. Indeed, the amount of memory in the computer is finite. If you try to allocate more memory than is available, then the function malloc returns the NULL-pointer to indicate that the allocation has failed. Therefore, be advised to always test whether malloc was successful or not.

So, a safer (and better) version of the function dynamicIntArray is:

```
int *dynamicIntArray(int sz) {
  int *ptr = malloc(sz*sizeof(int));
  if (ptr == NULL) {
    /* memory allocation failed */
    printf ("Error: memory allocation failed (out of memory?).\n");
    exit(-1); /* abort the program (with error code -1) */
  }
  return ptr;
}
```

Unlike ordinary (static) arrays, we need to deallocate space that we reserved using malloc once we do not need it any longer. If we do not do this, the amount of available memory will get exhausted. Memory is deallocated with the standard function free. If ptr is a pointer to a block of memory that we have allocated using malloc, then we can deallocate this memory as follows:

```
free(ptr);
```

Alert!

The function free deallocates the block of memory that ptr points to. However, the value of ptr itself is not changed (remember, C uses call-by-value). So, when the function free returns, we are left with a pointer ptr that points to an address that we are no longer allowed to use. It is a common misconceptoin that the function free would set ptr to NULL. If you want this, then you have to assign NULL to ptr yourself.

Note that we are only allowed to free pointers that point to dynamically allocated memory. The following uses of free are therefore wrong:

```
int a[100];
int *p = a;
free(a); /* This is wrong, a is allocated statically! */
free(p); /* This is wrong for the same reason. */
p = (int *)1000;
free(p); /* This is wrong too! */
```

Alert!

Note that pointers and static arrays are almost equivalent. The only difference between a pointer and a static array is in fact shown above: you cannot free a static chunk of memory (array), while you can free dynamically allocated memory that is assigned to a pointer.

In order to avoid wasting memory, it is necessary that each malloc is matched with exactly one corresponding free. If we omit one (or more) calls to the function free, memory is '*leaked*'. This is called a *memory leak*. On the other hand, if you accidentally free a pointer more than once, then the underlying memory management becomes inconsistent. This may result in very strange irreproducible errors (or bahviour) during execution of the program. Memory leaks and double-freeing of pointers are the type of (common) errors that are very hard to track down.

Of course, it is possible that we discover at some point that the size of a dynamically allocated array is insufficient. Using the combination of malloc and free, we can easily create a new dynamic array of the right size, copy the data and deallocate the memory of the old array. The following functions demonstrates this:

```
int *resizeDynamicIntArray(int oldSize, int *arr, int newSize) {
  int i, *ptr;
  /* allocate memory for the new array */
  ptr = dynamicIntArray(newSize);
  /* copy elements from old array to the new array */
  for (i=0; i < oldSize; i++) {
    ptr[i] = arr[i];
  }
  /* free memory of old array */
  free(arr);
  /* return new array */
  return ptr;
}
```

For example, if arr is a dynamic int array of size n, we can double its size with the call:

```
arr = resizeDynamicIntArray(n, arr, 2*n);
```

In practice, there is no real need to implement a function like resizeDynamicIntArray, since a similar functionality is already provided for by the standard C-library function realloc. Let ptr be a pointer that points to a chunk of memory of 10 **int**s. Now, we want to resize this dynamic array such that it can store 100 **int**s. We can do this using the following assignment:

```
ptr = realloc(ptr, 100*sizeof(int));
```

This assignment implements:

```
ptr = resizeDynamicIntArray(10, ptr, 100);
```

The function realloc returns a pointer to the new (resized) chunk of memory. Just like malloc, the function returns NULL on failure.

To conclude, here is a much better implementation of the function resizeDynamicIntArray:

```
int *resizeDynamicIntArray(int *arr, int newSize) {
  int *ptr = realloc(arr, newSize*sizeof(int));
  if (ptr == NULL) {
    /* memory allocation failed */
    printf ("Error: memory allocation failed (out of memory?).\n");
    exit(-1); /* abort the program (with error code -1) */
  }
  /* return new array */
  return ptr;
}
```

Note that this function lacks the formal parameter oldSize.

Now it is time to jump back to the original problem that we started with. We want to make a small program that reads in a series of integers that ends with the value zero. The task is to print the series of numbers in reverse order on the output.
This time, we make use of the dynamic memory allocation features discussed in this section:

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

int *resizeDynamicIntArray(int *arr, int newSize) {
  int *ptr = realloc(arr, newSize*sizeof(int));
  if (ptr == NULL) {
    /* memory allocation failed */
    printf ("Error: memory allocation failed (out of memory?).\n");
    exit(-1); /* abort the program (with error code -1) */
  }
  /* return new array */
  return ptr;
}

int inputSequence(int **seq) {
  int number, len = 0;
  *seq = NULL;
  do {
    scanf("%d", &number);
    if (number != 0) {
      *seq = resizeDynamicIntArray(*seq, len + 1);
      (*seq)[len] = number;
      len++;
    }
  } while (number != 0);
  return len;
}

void printSequence(int len, int seq[]) {
  while (len > 0) {
    len--;
    printf ("%d ", seq[len]);
  }
  printf ("\n");
}

int main(int argc, char *argv[]) {
  int *numbers;
  int length = inputSequence(&numbers);
  printSequence(length, numbers);
  free(numbers); /* deallocate array */
  return 0;
}
```

If you understand the theory of dynamic memory allocation well, then the above code fragment is straightforward. Still, it is a good exercise to have a closer look at the declaration

```
int inputSequence(int **seq)
```

The function inputSequence returns the number of integers on the input, so the return type is an **int** . More interesting is the formal parameter **seq. This is a double pointer, in other words a pointer to a pointer. This is necessary because we represent the sequence of numbers itself as a pointer (i.e. a dynamically allocated array), while we want to return this pointer to the caller. Since the parameter passing mechanism of C uses the call-by-value principle, it is therefore necessary to use an extra indirection in order to return the pointer. If this reasoning is not entirely clear to you, then you are strongly advised to study again the sections about pointers , call-by-value and call-by-reference in the previous chapter.

4.1.8 Strings: char arrays

A special variant of an array is the *string*, which is no more than an array of the type **char** that is used to store text (i.e. a sequence of characters). You can declare and initialise a string as follows:

```
char text[] = "Hello";
```

This creates a string (i.e. an array of type **char**) named text. The array is initialised with the text Hello. However, there is something special going on. The text Hello is only 5 characters long, but the array size is in fact 6 characters long. In C, the end of a string is indicated by the so-called 0-character. This is not the character '0', but the ASCII (i.e. integer) value 0: the ASCII value of the character '0' is 48. This terminating 0-character is called the *NUL-character*. Note that this differs from NULL, which is a pointer. Unfortunately, strings that are terminated by the NUL-character are still called *null-terminated strings* or *zero-terminated strings*.

Of course, we cannot type in or print the NUL-character, so a special notation for this character is required. Just like a new-line or tab, this character is escaped (notation using a \). The NUL -character is denoted \0. So, the above declaration is in fact the same as the following (clumsy) declaration:

```
char text[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

Alert!

A consequence of terminating strings with a NUL-character is that we need **char** arrays that have a size that is at least one character bigger than the text that we want to store in it. So, the compiler will not accept the following declaration:

```
char text[5] = "Hello"; /* wrong */
```

A correct declaration would be:

```
char text[6] = "Hello";
```

However, also correct is:

```
char text[10] = "Hello";
```

In the latter case the array still contains the text Hello, but we can extend it with 4 (not 5!) more characters. So, the first 5 elements of this array are filled with the characters 'H', 'e', 'l', 'l', 'o'. The next element (with index 5) contains the NUL-character. The remaining 4 elements contain arbitrary characters.

You can print a string on the screen using printf. The format-specifier for a string is %s. For example, the following code

```
printf("%s world!\n", text);
```

will produce the output

```
Hello world!
```

You can use scanf to read a string from the keyboard. However, beware! A string is an array, and hence a pointer! So, this time we must not prefix the name of the string with a pointer (since this would produce a double pointer). So, the correct way to read a string is:

```
scanf("%s", text);
```

The following code is wrong (the compiler accepts it, but the program crashes at runtime):

```
scanf("%s", &text); /* wrong! */
```

In the standard C library many handy functions are present that operate on strings. If you want to use these functions, you must include the file `string.h` at the beginning of your program.

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

We will discuss a few of these functions. We start with the function strlen:

```
int strlen(char *s);
```

The function strlen returns the length of its string argument s, not including the NUL-character. So, the following statement will produce the output 5 (not 6, which is the size of the string text):

```
char text[] = "Hello";

printf("%d\n", strlen(text));
```

It is not possible to assign a string to another string by a simple assignment statement, since this would be a pointer assignment (see the discussion on pointer assignments in Section 4.1.5 on page 51). For this reason, there is the function strcpy, which copies a string.

```
char *strcpy(char *dest, char *src);
```

The function strcpy copies the string src into the string dest (including the terminating NUL-character). Of course, the (array)size of the string dest must be at least as large as the size of src. The function returns the value of the pointer dest, but this is rarely used: this return value allows us, for example, to implement two string copies in a single statement:

```
strcpy(dest1, strcpy(dest0, src)); /* src is copied into dest0 and dest1 */
```

The function strcat is used to extend a string.

```
char *strcat(char *dest, char *src);
```

The function strcat appends the string src to the string dest, overwriting the terminating NUL-character at the end of dest, and then adds a terminating NUL-character. The strings dest and src may not overlap, and the string dest must have enough space for the result. This operation is commonly called *concatenation* of strings. For the same reasons as strcpy, the function returns the pointer to dest. So, the following code fragment is a complicated way to print the text "Hello world!" on the screen twice:

```
char hello[13] = "Hello";
char world[8] = " ";
printf("%s\n", strcat(hello, strcat(world, "world!")));
printf("%s\n", hello);
```

Comparing strings using the standard comparison operators ==, !=, <, <=, > and >= is almost always useless. The reason is that if we compare two strings directly, then we compare in fact two pointers (hence memory addresses), while you probably wanted to test whether the contents of the strings are the same. So, the following code fragment

```
char a[] = "hello";
char b[] = "hello";
char *c = a;

printf("%s\n", (a == b ? "Equal" : "Not equal"));
printf("%s\n", (a == c ? "Equal" : "Not equal"));
printf("%s\n", (b == c ? "Equal" : "Not equal"));
```

will produce the (probably unwanted) output:

```
Not Equal
Equal
Not Equal
```

If we want to compare the contents of two strings, then we should use the function strcmp.

```
int strcmp(char *s0, char *s1);
```

The function strcmp compares the two strings s0 and s1. It returns an integer less than, equal to, or greater than zero if s0 is found, respectively, to be less than, equal, or greater than s1. Here 'less than' means that if we put the strings s0 and s1 in alphabetical order, then s0 will come first. So, the following code fragment

```
char s0[]="elephant";
char s1[]="fly";
if (strcmp(s0,s1) == 0) {
  printf ("The strings are equal\n");
} else {
  if (strcmp(s0,s1) < 0) {
    printf ("The string '%s' is less than '%s'\n", s0, s1);
  } else {
    printf ("The string '%s' is greater than '%s'\n", s0, s1);
  }
}
```

will produce the output

```
The string 'elephant' is less than 'fly'
```

To conclude this chapter, here is a small variation on the standard "hello world"-program using strings and dynamic memory allocation. Try to figure out the logic of the program yourself.

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

char *readString() {
  char longString[1000];
  char *string;
  scanf("%s", longString);
  string = malloc((1+strlen(longString))*sizeof(char));
  strcpy(string, longString);
  return string;
}

int main(int argc, char *argv[]) {
  char *name;
  printf("What is your name? ");
  name = readString();
  printf("Hello %s, how are you?\n", name);
  free(name);
  return 0;
}
```

4.2 Tutorial exercises

In the following eight exercises, let a be declared as follows:

```
    int a[100];
```

4.2.1 Maximum

Write a program fragment that computes the maximum value in the array a.

4.2.2 Maxima

Write a program fragment that counts the number of indices k where a[k] is maximal.

4.2.3 Related

We call two indices p and q *related* if and only if a[p] == q and a[q] == p.
Print a list of all related pairs (p,q) for which p<=q.

4.2.4 Steps

Write a program fragment that counts the number of indices k for which a[k] < a[k+1].

4.2.5 Longest increasing subsequence

Write a program fragment that computes the length of the longest increasing (sub)sequence in the array a.

4.2.6 Factorial

Write a program fragment that assigns the factorial of k (i.e. k!) to a[k] for every valid index k of the array a. You do not have to worry about overflow.

4.2.7 Histogram

Assume that the array a contains only decimal digits. Count for every digit k the number of occurences of the digit k in the array a and store it in h[k].

4.2.8 Big integer

The programming language C has four built-in integer types: **char**, **short**, **int**, and **long**. Clearly, the type **long** has the largest range of values that it can represent. On a standard PC, the largest value that we can store in an **unsigned long** is $2^{64} - 1 = 18446744073709551615$.

However, if we want to do arithmetic with integers that are even greater than this value, we will have to built our own big integers. A natural choice for the representation of a big integer is a series of digits (which we store in the array a). We interpret the array a as the decimal representation of a big integer n:

$$n = \mathsf{a}[99] * 10^{99} + \mathsf{a}[98] * 10^{98} + ... + \mathsf{a}[2] * 10^2 + \mathsf{a}[1] * 10 + \mathsf{a}[0]$$

Write a program fragment that computes 2*n. The result must be stored again in a using the above representation. You do not have to worry about overflowing the number 10^{100}.

4.2.9 Water management

We consider a square undulating terrain with a fairly permeable soil structure. This terrain is represented by means of a two-dimensional array:

```
int height[120][80];
```

where height[i][j] denotes the height of the terrain at the coordinate (i, j).

When the water level rises, the area gets partially submerged. The number of grid points with a height smaller than the water level is a measure of the area of the flooded land. Write a program fragment that, given the array height and the water level, computes the number of flooded points.

4.2.10 Pascal's triangle

Given is an initial part of Pascal's triangle:

```
1
1  1
1  2   1
1  3   3   1
1  4   6   4   1
1  5  10  10   5   1
```

In Pascal's triangle, each number (not equal to one) is the sum of its upper and upper-left neighbour. Write a program that stores the first 15 lines of Pascal's triangle in a two-dimensional array. The program must also print these lines of the triangle on the screen.

Chapter 5

Recursion

Recursion is recursion is recursion is recursion is recursion is recursion is recursion ...

5.1 Recursion

Recursion is the phenomenon that a function is defined in terms of itself (in other words, it calls itself). There is actually no difference between a recursive function call and a regular one. Recursion is not a new language construct. It is simply an application of what you have already learned about function calls. However, it does require a new style of thinking about solving problems. The key is that you do not solve a problem directly, but you reduce the problem to a smaller instance of it. In the beginning, this style of thinking can be confusing. Once you get used to it, you will see that it results in very nice, compact and elegant algorithms that solve complicated problems much more easily than non-recursive algorithms.

Recursive reasoning is actually quite common in mathematics. For example, the set of *natural numbers* is recursively defined as follows:

- 0 is a natural number.
- x is a natural number if $x - 1$ is a natural number (for $x > 0$).

The second case in this definition is recursive: it defines a natural number in terms of a natural number!

Often, recusion is a natural and elegant way to define (mathematical) functions. However, you need to be careful. If you implement recursive functions in a programming language, then you need to make sure that the function terminates, i.e. it should not call itself indefinitely. For example, the following function f on natural numbers is recursive, but not well defined:

$$f(n) = f(n+1) - 1$$

The problem is that the function iterates forever, without getting any closer to an answer. However, if we extend the definition with a *base* case, then the definition is perfectly fine:

$$f(0) = 0, \qquad f(n) = f(n+1) - 1$$

Note that the function f is actually the *identity function*, i.e. the function $f(n) = n$.

In general, we prefer to define a recursive function in terms of a smaller case (i.e. smaller value for the argument(s)). Therefore, we transform, using simple arithmetic, the above definition in the following equivalent (and preferred) one:

$$f(0) = 0, \qquad f(n+1) = f(n) + 1$$

In C, you can implement recursive functions quite naturally since C allows self-references of functions. Again, in actual implementations of recursive functions, you need to take care that the functions terminate.

For *each* recursive call of a function, the compiler generates code that reserves memory for all local variables and parameters of the function. These variables are stored in a piece of memory that is called the *stack*. The programmer does not need to worry about the stack and allocation of memory from the stack. This is all handled by the compiler. However, the stack is limited in size, so indefinite (or very deep) recursion will run out of stack space and the program will crash: we call this a *stack overflow*. By carefully designing our recursive functions, and always checking whether we will eventually reach the base case(s), we can avoid this.

Alert!

Each problem that can be solved iteratively (for example, using a **while** loop), can also be solved using recursion. The converse is also true: each recursive algorithm can be converted in an iterative algorithm. However, converting a recursive algorithm into an iterative algorithm can be complicated since it may be necessary to implement (simulate) the stack yourself.
In general, if we need to choose between a recursive and an iterative algorithm, the iterative version has preference provided that the iterative code is not (much) more complicated than the recursive one. There are two main reasons for this: an iterative algorithm is usually faster and there is no danger of stack overflow.
On the other hand, if a recursive algorithm is clearly more elegant and easier than an iterative one, then it is certainly a good idea to go for the recursive one. In the remainder of this chapter, we will focus on recursive functions.

5.1.1 Example: recursive factorial function

To make the discussion on recursion a little less abstract let us have a look at a concrete example. The factorial of a natural number is recursively defined as follows:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \qquad \text{for a natural number } n > 0 \end{aligned}$$

In a recursive function we always distinguish between one or more *base cases* and one or more *recursive cases*. The base cases are the cases where the answer is given directly. Usually, the base case(s) is(are) given for the smallest possible arguments of the function. The recursive definition of the factorial clearly has one base case: $0! = 1$.

The recursive cases require at least one recursive step to compute the answer. For a correct recursive definition, it is necessary that the arguments in the recursive step are always smaller in some sense than the arguments in the current step. Here, smaller need not be smaller in the numerical sense, but rather 'closer to one of the base cases'.
For example, the computation of 6! reduces to the computation of 5!, where $n = 5$ is closer to the base case $n = 0$ than the original argument $n = 6$. By recursively applying the definition we can compute 6! as follows:

$$\begin{aligned} 6! &= 6 \times 5! \\ &= 6 \times 5 \times 4! \\ &= 6 \times 5 \times 4 \times 3! \\ &= 6 \times 5 \times 4 \times 3 \times 2! \\ &= 6 \times 5 \times 4 \times 3 \times 2 \times 1! \\ &= 6 \times 5 \times 4 \times 3 \times 2 \times 1 \times 0! \\ &= 6 \times 5 \times 4 \times 3 \times 2 \times 1 \times 1 = 720 \end{aligned}$$

A recursive implementation of the factorial in C resembles closely the above definition:

```c
int fac(int n) {
  if (n == 0) { /* base case */
    return 1;
  }
  /* recursive case */
  return n*fac(n-1);
}
```

Of course, it is easy to implement an iterative factorial function.

```c
int fac(int n) {
  int f = 1;
  while (n > 0) {
    f *= n;
    n--;
  }
  return f;
}
```

Which implementation is better, is a matter of taste. The recursive function may be a bit slower, since stack operations are needed for each recursive function call. On the other hand, modern compilers are very good in optimizing recursive code, and it is very likely that the compiler will convert the recursive function into iterative machine code. In that case you will not notice any performance penalty for the recursive implementation.

5.1.2 Another example: Fibonacci

The Fibonacci sequence is named after Leonardo Fibonacci, also known as Leonardo of Pisa (around 1170 - 1250 AD). He introduced the sequence as a mathematical model for the growth of a rabbit population. The sequence is derived from the following assumptions:

- We start with a pair of young rabbits.
- A pair of rabbits is mature after one year.
- Every year, a mature pair of rabbits produces offspring: one young pair of rabbits.
- Rabbits do not die.

Verify yourself that the number of mature rabbit pairs per year is given by the following sequence : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

We introduce the notation $F(n)$ for the number of mature rabbit pairs in year n. It is not hard to find out that the function $F(n)$ is given by the following recurrence:

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n) &= F(n-2) + F(n-1) \qquad \text{for a natural number } n > 1 \end{aligned}$$

Clearly, the recursion for $F(n)$ has two base cases: the cases $n = 0$ and $n = 1$. In both cases, the answer is given directly.

We need two base cases as a result of the structure of the recursive case. In the recursive case, to compute $F(n)$, we need the values of $F(n-1)$ and $F(n)$. Hence, to make sure that the recursion terminates, we need two consecutive base cases: an odd and an even base case. Convince yourself that the recursion does not terminate if we omit one of the base cases.

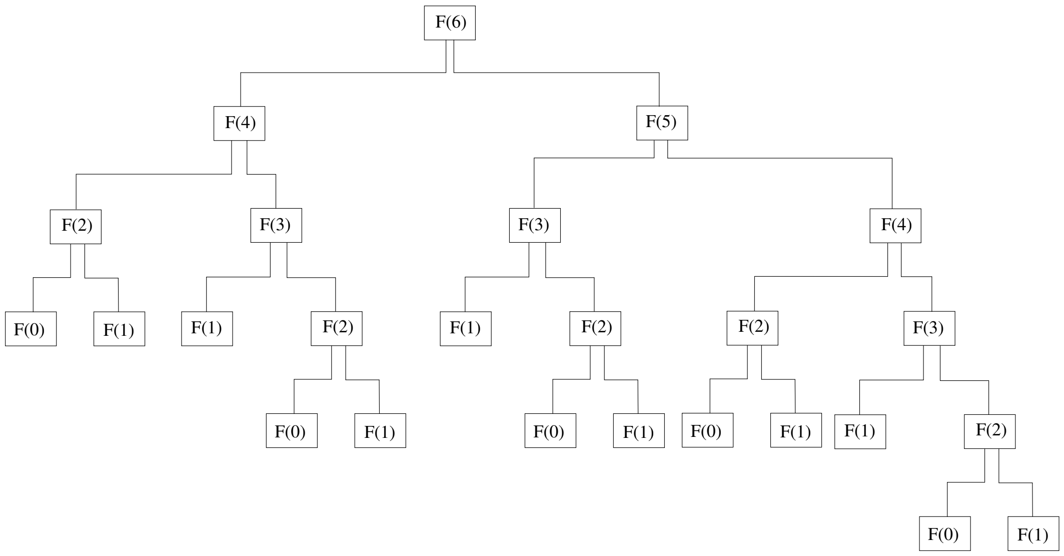


Figure 5.1: Recursion tree of $F(6)$

It is easy to implement the fibonacci function recursively in C:

```
int fibonacci(int n) {
    /* 2 base cases: F(0)=0 and F(1)=1 */
    if ((n == 0) || (n == 1)) {
      return n;
    }
    /* recursive case */
    return fibonacci(n−2) + fibonacci(n−1);
}
```

5.2 Dynamic programming

The recursive function fibonacci may be a direct and elegant implementation of the recursive function $F(n)$, but it is quite inefficient. The function computes many function values several times. For example, the computation of fibonacci(6) reduces to the recursive computation of fibonacci(4) and fibonacci(5). The latter computation, on its turn, needs fibonnaci(3) and fibonacci (4). So, we need to compute fibonacci(4) twice. In fact, it is even worse. This becomes clear if we take a look at Fig. 5.1. For the computation of fibonacci(6), we need to compute fibonacci(5) once, fibonacci(4) twice, fibonacci(3) three times, and eventually fibonacci(2) 5 times. Moreover, we reach a base case 13 times. Altogether, we need to call the function fibonacci 25 times, which is clear from the observation that the recursion tree in Fig. 5.1 has 25 nodes.

We can avoid spurious computations using a technique called *dynamic programming*. The idea is very simple. We use *memorisation* to avoid repeated calculation of (intermediate) results. We reserve some additional memory space for the storage of (intermediate) results. As soon as we discover that we try to determine some previously computed value, we reuse the stored result.

The largest value from the Fibonacci sequence that fits in a 32-bit (signed) **int** is $F(46) = 1836311903$. So, we introduce an extra **int** array fibmem with a size of 47 elements. We declare this array globally, such that it is automatically initialised with zeroes. This feature comes in handy: if an element has the value zero, it means that it was not assigned a value yet. The following code fragment shows a recursive implementation of the function fibonacci that makes use of dynamic programming:

```
int fibmem[47]; /* global array, so it is initialized with zeroes */

int fibonacci(int n) {
    /* 2 base cases: F(0)=0 and F(1)=1 */
    if ((n == 0) || (n == 1)) {
      return n;
    }
    /* recursive case */
    if (fibmem[n] == 0) { /* result unknown yet */
      fibmem[n] = fibonacci(n-2) + fibonacci(n-1); /* recursive calls */
    }
    /* here: fibmem[n] == F(n) */
    return fibmem[n]; /*
}
```

To compare the performance of the two algorithms, you are advised to implement both recursive versions of the fibonacci function on a PC and test it. You can try yourself to determine $F(46)$ using both approaches. Using the first algorithm (without memorisation), you will have to wait approximately a minute until you get an answer. The second version will last no more than a few milliseconds. A huge improvement!

5.3 Towers of Hanoi

The '*towers of Hanoi*' is a puzzle which was first introduced by the French mathematician Édouard Lucas in 1883. Rumours are that Lucas was inspired by a legend about an Indian temple in Kashi Vishwanath which contains a large room with three rods in it surrounded by 64 golden discs. Brahmin priests, acting out the command of an ancient prophecy, have been moving these discs, in accordance with the rules of the Brahma, since that time. The puzzle is therefore also known as the Tower of Brahma puzzle. According to the legend, when the last move of the puzzle is completed, the world will end.

The game that Lucas introduced in 1883 consists of a board containing three rods. At the start of the game, a pyramidal tower of discs (with a hole in the middle) is placed on one of the rods. Each disc has a different diameter, and the discs are placed so that the smallest is on top and the largest at the bottom (see Fig. 5.2).

The objective of the puzzle is to move the entire pyramidal stack of discs to another rod, obeying the following rules:

- Only one disc is moved at a time.
- Each move consists of taking the upper disc from one of the rods and sliding it onto another rod, on top of the other discs that may already be present on that rod.
- No disc may be placed on top of a smaller disc.

The key to solving this puzzle is to recognize that it can be solved by breaking the problem down into a collection of smaller problems and further breaking those problems down into even smaller problems until a solution is reached. For example, assume that we have a solution to the problem with $n-1$ discs (where $n > 0$). Then, we can solve the problem with n discs in the following three steps:

1. Move the upper $n-1$ discs to an empty rod, using the solution for the problem with $n-1$ discs. At the end of this process you are left with one empty rod, a rod with the largest disc, and a rod with a pyramid of $n-1$ discs.
2. Move the largest disc to the empty rod.

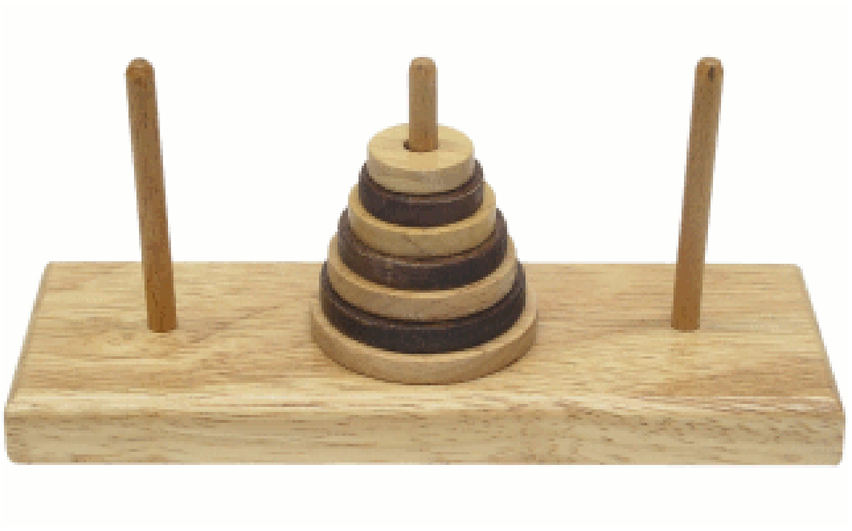


Figure 5.2: Initial configuration of the 'tower of Hanoi'.

3. Move the pyramid of $n-1$ discs on top of the largest disc using the solution procedure for $n-1$ discs again.

We can translate this procedure into a recursive function hanoi with the following prototype:

```
void hanoi(int numberOfDiscs, int src, int dest);
```

The formal parameters of this function play the following roles:

- **int** numberOfDiscs: The number of discs that (still) need to be moved.
- **int** src: The number of the rod (source) that (at the start of the function) contains the pyramid with numberOfDiscs discs.
- **int** dest: The number of the rod (destination) that we want to move the pyramid to.

We enumerate the rods 1, 2, and 3. So, the function call hanoi(5,1,3) should solve the problem of moving 5 discs from rod 1 to rod 3. Note that we can use rod 2 when we move discs from rod 1 to rod 3. In general, as a result of the chosen numbering of the rods, we can say that we can use the rod with number 6−src−dest when we move discs from src to dest. So, it is not necessary to have a third parameter **int** via, since we can easily compute it given src and dest using the assignment via = 6 − src − dest.

We first consider the base case of the recursion. In general, the base case should be the smallest case that we can think of. In the above informal description, the solution is even correct for $n = 1$, wherein use is made of the solution to $n = 0$. So we choose as the base case the situation with $n = 0$. Moving zero discs is a dummy operation, so in the base case of this recursion the function returns directly.

It follows from the informal description of the solution procedure that the recursive case consists of three steps. First we move the upper numberOfDiscs−1 from the rod src to the rod via. Since this is a smaller version of the original problem (i.e. one disc fewer to move), we may use the assumption that our recursive function already works for smaller cases. In other words, we can implement the first step using the recursive function call:

```
    hanoi(numberOfDiscs - 1, src, via);
```

After this function call, we move the (remaining) largest disc from rod src to rod dest and print this move on the screen. In the last step, we need to move the pyramid with numberOfDiscs − 1 discs from rod via to the rod dest. Following the same reasoning as in the first step, we can do this using the following recursive function call:

```
    hanoi(numberOfDiscs − 1, via, dest);
```

The complete program is short and elegant:

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

void hanoi(int numberOfDiscs, int src, int dest) {
  /* base case: numberOfDiscs == 0, there is nothing to do */
  if (numberOfDiscs != 0) {
    /* recursive case */
    int via = 6 − src − dest;
    hanoi(numberOfDiscs − 1, src, via);
    printf ("Move disc from rod %d to rod %d.\n", src, dest);
    hanoi(numberOfDiscs − 1, via, dest);
  }
}

int main (int argc, char *argv[]) {
  int numDiscs;
  printf("Number of discs? ");
  scanf("%d", &numDiscs);
  hanoi(numDiscs, 1, 3);
  return 0;
}
```

What remains is the answer to the question when the world will end. If you run the above program for a problem with five discs, then the solution is almost immediately printed on your screen. However, if you try to run the program for 64 discs then you will notice that the program appears to produce output forever. In fact, your computer will certainly be defective before the output stops (i.e. before the problem is solved).

The following analysis explains this. Let $T(n)$ be the number of moves that we need to make to move a pyramid of n (where $n \geq 0$) discs. It is clear that $T(0) = 0$. For some $n > 0$, we use two recursive function calls that each make $T(n-1)$ moves. Moreover, we move the largest disc in between. So, we conclude that $T(n) = 1 + 2T(n-1)$. Let us have a look at a few elements of this sequence:

$$0, 1, 3, 7, 15, 31, 63, 127, \ldots$$

We recognize the pattern $T(n) = 2^n - 1$. This is indeed the right formula, which is easily proved using mathematical induction (try this yourself). We are interested in the value of $T(64)$. It is an incredibly large number, $T(64) = 18446744073709551615$. Now, suppose that your PC can perform one billion moves per second (which is overly optimistic) then it would take 584942 years to solve the 64-discs problem. Priests are much slower, so we need not be very concerned about the end of time.

5.4 The eight queens problem

Another puzzle that can be solved nicely using recursion is the famous *eight queens problem*. One of the pieces in the game of chess is the queen. A queen can move from a square (row,column) to another square of the board provided that this square is in the same row, same column or same diagonal as the square (row,column). The objective of the eight queens puzzle is to place eight queens on a chessboard in such a way that no queen can reach (capture) another in a single move. In other words, a solution requires that no two queens share the same row, column, or diagonal.

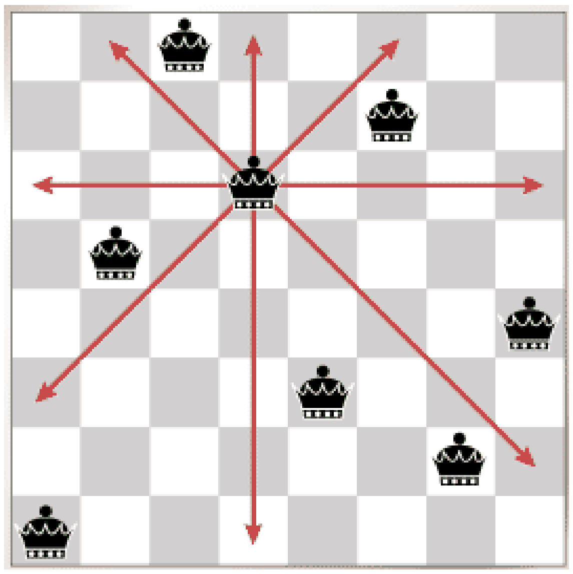


Figure 5.3: A solution to the 8 queens problem.

The eight queens problem has 92 solutions. However, you could argue that there are only 12 unique solutions since some of these 92 solutions differ only by rotations and reflections of the board. One of the solutions is shown in Fig. 5.3, including a pictorial description of the moves that a queen can make.

If you try to solve this puzzle yourself using a chessboard and eight queens (use the pawns, since there are only two queens in a chess set), then you will find that is not easy to find a solution quickly, as there are 4,426,165,368 (i.e. the number of ways to choose 8 squares from 64 squares) possible arrangements of eight queens on a chess board. However, with a recursive approach, the problem is solved easily. In fact, the recursive function that we will design in this section will generate all 92 solutions in a few milliseconds on a standard PC.

A chessboard has 64 squares: eight rows of eight columns. Unlike a real chessboard, we number the rows and columns 0 to 7. In a solution arrangement, there can be no two queens in the same row. So, we can reformulate the problem as follows: *Place on each row a queen such that no two queens share the same column, or diagonal.*

A consequence of this reformulation is that we can represent a board arrangement with a simple one-dimensional array pos (from position):

```
int pos[8];
```

We interpret the array as follows: pos[row] yields the column of the position of the queen that is placed in the row row, i.e. the location of the queen is (row,pos[row]). Using this representation, the solution from Fig. 5.3, where the lower left square has the coordinate (0,0), would be the array pos[]={0,6,4,7,1,3,5,2}.

We try to solve the problem systematically: first we try to place a queen in row 0, followed by a queen in row 1, and so on. Of course, if we try to place a queen in row r, then we must place it in a column that is not in conflict with the queens that we have already placed in the rows from $[0..r)$. If such a column does not exist, we conclude that we have made a wrong choice in the placement of a queen in one (or more) of the previous rows. In this case we need to jump back to the last choice we made, and make another choice if possible. If there is no other choice, then we jump back even further, and so on. This jumping back is called *backtracking* in the literature on algorithms and data structures. It is a very general method for solving problems by searching for solutions (like the eight queens problem).

The above description suggests a recursive solution, where the recursion is on the row numbers:

after placing a queen in row r we try recursively to place a queen in row r+1. Note that, although the value of r grows, this is a valid recursion: the recursion stops when we reach r==8 which is the base case of the recursion. So, by incrementing r, we get closer to the solution.

We introduce the function placeQueen with the prototype:

```
void placeQueen(int row, int pos[8]);
```

As noted, the base case is reached when row==8. In this case, we print the solution on the screen using the function printPosition.

```
void printPosition(int pos[8]) {
  int i;
  printf ("%d", pos[0]);
  for (i=1; i < 8; i++) {
    printf (" %d", pos[i]);
  }
  printf ("\n");
}
```

The function printPosition prints the array pos, i.e. a sequence of column numbers, on the screen. It is left as an exercise to change its body such that it produces an output like:

```
. Q . . . . . .
. . . . . Q . .
. . . . . . . Q
. . Q . . . . .
Q . . . . . . .
. . . Q . . . .
. . . . . . Q .
. . . . Q . . .
```

The recursive case is the situation where there are already non-conflicting queens placed on rows r with r<row<8. So, we need to find a column number column which is not already occupied by a previously placed queen.

If such a column exists, then we need to check that the corresponding diagonals are not in conflict with other queens. If this requirement is met as well, then we place a queen on the location (row,column) and go on using a recursive call with the argument row+1.

If a non-conflicting column does not exist, then the function terminates and returns to the caller, i.e. the previous recursive function call at the level row−1. At that level, the loop that searches for a suitable column will resume, and tries to find another column for that recursion level. The reason that the loop is able to resume lies in the fact that the compiler stores the local variables of each recursion level on the stack. When the recursion at some level terminates, we fall back to the previous level, and all variables of that level are unmodified (they are saved on the stack). In other words, we do not have to implement backtracking explicitly! The stack takes care of that. Backtracking is a direct consequence of the chosen recursive solution method.
The complete code that solves the eight queens problem is given in the following program:

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

#define ABS(a) ((a)<0 ? (−(a)) : (a))

void printPosition(int pos[8]) {
  int i;
  printf ("%d", pos[0]);
  for (i=1; i < 8; i++) {
    printf (" %d", pos[i]);
  }
  printf ("\n");
}

void placeQueen(int row, int pos[8]) {
  if (row == 8) {
    /* base case */
    printPosition(pos);
  } else {
    /* recursive case */
    int column;
    for (column=0; column < 8; column++) {
      int r;
      for (r=0; r < row; r++) {
        if ((pos[r] == column) /* queen in this column? */
           || (ABS(pos[r]-column) == row-r)) /* queen on diagonal? */ {
          break;
        }
      }
      if (r==row) { /* square is not attacked => place queen */
        pos[row] = column;
        placeQueen(row+1, pos);
      }
    }
  }
}

int main(int argc, char *argv[]) {
  int pos[8];
  placeQueen(0, pos);
  return 0;
}
```

5.5 A backtracking strategy for solving a Sudoku

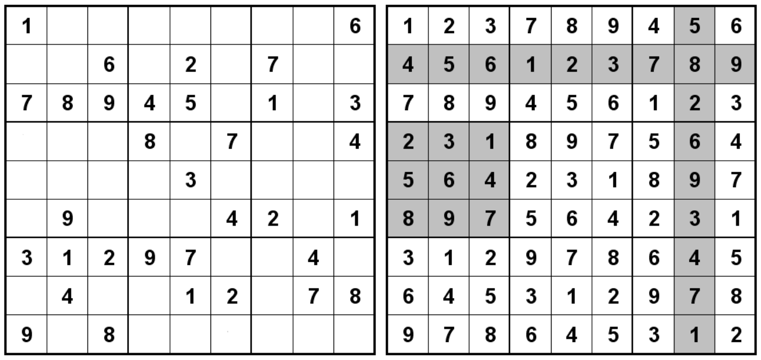
A sudoku is a puzzle on a 9×9 grid. The objective is to fill the grid with digits so that each column, each row, and each of the nine 3×3 sub-grids (also called "blocks") contains all digits from 1 to 9. Initially, the puzzle consists of a partially completed grid. A correct sudoku has a unique solution. An example of a sudoku and its solution is given in Fig. 5.4.

At a first glance, solving a sudoku has little in common with solving the eight queens problem from section 5.4. Yet, the algorithm that we will develop for solving a sudoku uses the same technique that we used for solving the eight queens problem, namely backtracking.

The key lies in the recognition of the recursive nature of sudokus. The base case is a completed sudoku, i.e. a sudoku with no empty squares left. In this case we need not do anything. If there are still empty squares left (i.e. the recursive case) then we can try to fill in a digit (at some empty square) and we are left with a new sudoku with one empty square fewer than the original problem. In other words, we have a smaller instance of the original problem. Of course, it is possible that at some point we fill in a wrong digit, but this is solved again by backtracking.

A natural representation of a sudoku is a two-dimensional array of **int**s:

```
int sudoku[9][9];
```



1								6
		6		2		7		
7	8	9	4	5		1		3
			8		7			4
				3				
	9				4	2		1
3	1	2	9	7			4	
	4			1	2		7	8
9		8						

1	2	3	7	8	9	4	5	6
4	5	6	1	2	3	7	8	9
7	8	9	4	5	6	1	2	3
2	3	1	8	9	7	5	6	4
5	6	4	2	3	1	8	9	7
8	9	7	5	6	4	2	3	1
3	1	2	9	7	8	6	4	5
6	4	5	3	1	2	9	7	8
9	7	8	6	4	5	3	1	2

Figure 5.4: A sudoku (left) and its solution (right). Each row, each column, and each block contains all of the digits from 1 to 9.

We decide that the input of our program is coded as follows: 9 lines containing 9 digits each. An empty square is denoted with the digit 0. So, the input that corresponds to Fig. 5.4 would be:

```
100000006
006020700
789450103
000807004
000030000
090004201
312970040
040012078
908000000
```

We initialise the array sudoku using the function readSudoku.

```
void readSudoku(int sudoku[9][9]) {
  int row, column;
  for (row=0; row < 9; row++) {
    for (column=0; column < 9; column++) {
      sudoku[row][column] = getchar() − '0';
    }
    getchar(); /* skip newline (\n) character */
  }
}
```

Of course, we also want to output the solution. We choose the following output format, which is easier to read than the input format:

```
+---+---+---+
|123|789|456|
|456|123|789|
|789|456|123|
+---+---+---+
|231|897|564|
|564|231|897|
|897|564|231|
+---+---+---+
|312|978|645|
|645|312|978|
|978|645|312|
+---+---+---+
```

The function printSudoku produces output in the above format (check the code yourself):

```
void printSudoku(int sudoku[9][9]) {
  int row, column;
  for (row=0; row < 9; row++) {
    if (row%3 == 0) {
      printf("+---+---+---+\n");
    }
    for (column=0; column < 9; column++) {
      if (column%3 == 0) {
        printf("|");
      }
      printf ("%d", sudoku[row][column]);
    }
    printf ("|\n");
  }
  printf("+---+---+---+\n");
}
```

We need a function that checks whether a given digit is allowed on a square with coordinate (row,column). The digit is allowed at that location if it is not already present in the corresponding row, column, or block. We use the function digitPossible to perform this check. It returns FALSE if we cannot place digit at location (row,column), while it returns TRUE otherwise. Again, please check the logic of the code of this function yourself. Especially the code that checks whether the digit already occurs in the same block is instructive.

```
#define FALSE 0
#define TRUE 1

int digitPossible(int row, int column, int digit, int sudoku[9][9]) {
  int i, r, r0, r1, c, c0, c1;
  /* digit in same row or column? */
  for (i=0; i < 9; i++) {
    if ((sudoku[row][i] == digit) || (sudoku[i][column] == digit)) {
      return FALSE;
    }
  }
  /* digit in same 3x3 block? */
  r0 = row - row%3;
  r1 = r0 + 3;
  c0 = column - column%3;
  c1 = c0 + 3;
  for (r=r0; r<r1; r++) {
    for (c=c0; c<c1; c++) {
      if (sudoku[r][c] == digit) {
        return FALSE;
      }
    }
  }
  return TRUE;
}
```

Now that we have all ingredients available, we focus on solving sudokus recursively. We introduce the function solveSudoku with the following prototype:

```
void solveSudoku(int row, int column, int sudoku[9][9]);
```

This function tries to fill in a suitable digit in the square with coordinates (row,column). If it finds a suitable digit, it will recurse with the coordinates of the next square (which is the neighbouring square (row,column+1), or the square (row+1,0) in case we reached the right margin of the sudoku). Given the square with the coordinates (row,column), we find the next square (r,c) with the following code:

```
if (column < 8) {
  /* Next square is (row,column+1) */
  r = row;
  c = column+1;
} else {
  /* Next square is (row+1,0) */
  r = row+1;
  c = 0;
}
```

The base case of the recursion is the case in which the entire sudoku is filled by the recursive solution strategy, i.e. when row==9. In that case, we print the solution on the screen.

```
if (row == 9) {
  printSudoku(sudoku);
  return;
}
```

In the recursive case we first need to check whether there is already a digit in the square (row, column). If this is the case, then we can recurse immediately.

```
if (sudoku[row][column] != 0) {
  /* There is already a digit here, continue */
  solveSudoku(r, c, sudoku);
  return;
}
```

In case the square (row,column) is empty, we try to fill in each possible digit and enter the recursion. This approach is very similar to the solution of the eight queens problem. However, this time there is a small subtlety: when we return from the recursion, we need to remove the digit again. In fact, if you solve a sudoku with pen and pencil, you do the same thing. Once you have discovered that you tried to fill in a wrong digit, you erase all digits that were based on the assumption that the digit was correct. We arrive at the following code fragment:

```
/* Empty square. Try all possible digits */
for (digit=1; digit<10; digit++) {
  if (digitPossible(row, column, digit, sudoku)) {
    /* fill in digit, and recurse */
    sudoku[row][column] = digit;
    solveSudoku(r, c, sudoku);
    /* clear square */
    sudoku[row][column] = 0;
  }
}
```

This concludes our analysis. Now that we have finished the program, it is a good moment to compare the program with the one that solves the eight queens problem (section 5.4). You will see that the structure of the two programs is very similar.

For the sake of completeness, here is the complete sudoku program:

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

#define FALSE 0
#define TRUE 1

void readSudoku(int sudoku[9][9]) {
  int row, column;
  for (row=0; row < 9; row++) {
    for (column=0; column < 9; column++) {
      sudoku[row][column] = getchar() - '0';
    }
    getchar(); /* skip newline (\n) character */
  }
}

void printSudoku(int sudoku[9][9]) {
  int row, column;
  for (row=0; row < 9; row++) {
    if (row%3 == 0) {
      printf("+---+---+---+\n");
    }
    for (column=0; column < 9; column++) {
      if (column%3 == 0) {
        printf("|");
      }
      printf ("%d", sudoku[row][column]);
    }
    printf ("|\n");
  }
  printf("+---+---+---+\n");
}

int digitPossible(int row, int column, int digit, int sudoku[9][9]) {
  int i, r, r0, r1, c, c0, c1;
  /* digit in same row or column? */
  for (i=0; i < 9; i++) {
    if ((sudoku[row][i] == digit) || (sudoku[i][column] == digit)) {
      return FALSE;
    }
  }
  /* digit in same 3x3 block? */
  r0 = row - row%3;
  r1 = r0 + 3;
  c0 = column - column%3;
  c1 = c0 + 3;
  for (r=r0; r<r1; r++) {
    for (c=c0; c<c1; c++) {
      if (sudoku[r][c] == digit) {
        return FALSE;
      }
    }
  }
  return TRUE;
}

void solveSudoku(int row, int column, int sudoku[9][9]) {
  int digit, r, c;
  /* base case */
  if (row == 9) {
    printSudoku(sudoku);
    return;
  }
  /* recursive case */
  if (column < 8) {
    /* Next square is (row,column+1) */
    r = row;
    c = column+1;
  } else {
    /* Next square is (row+1,0) */
    r = row+1;
    c = 0;
  }
  if (sudoku[row][column] != 0) {
    /* There is already a digit here, continue */
    solveSudoku(r, c, sudoku);
    return;
  }
  /* Empty square. Try all possible digits */
  for (digit=1; digit<10; digit++) {
    if (digitPossible(row, column, digit, sudoku)) {
      /* fill in digit, and recurse */
      sudoku[row][column] = digit;
      solveSudoku(r, c, sudoku);
      /* clear square */
      sudoku[row][column] = 0;
    }
  }
}

int main(int argc, char *argv[]) {
  int sudoku[9][9];
  readSudoku(sudoku);
  solveSudoku(0, 0, sudoku);
  return 0;
}
```

5.6 Tutorial exercises

5.6.1 Segment sum

Write a recursive function segmentSum that implements the following specification:

```
int segmentSum(int length, int seq[], int left, int right) {
    // PRE: 0 <= left <= right <= length
    // RETURN: (SUM i : left <= i < right : seq[i])
}
```

5.6.2 Segment maximum

Write a recursive function segmentMax that implements the following specification:

```
int segmentMax(int length, int seq[], int left, int right) {
    // PRE: 0 <= left <= right <= length
    // RETURN: (MAX i : left <= i < right : seq[i])
}
```

5.6.3 Number of ones in binary representation

Write a recursive function oneBits(**unsigned int** n) that returns the number of one-bits in the binary representation of the positive integer n. For example, the binary representation of 42 is 00101010, so oneBits(42) should return 3.

5.6.4 Number of subsequences with given sum/product

Given is an array arr with 100 integers.

```
int arr[100];
```

A *subsequence* of arr is a not necessarily contiguous sequence of elements taken from the array arr. Note that this differs from a set, since an element may occur more than once. Write a recursive function subSum that computes the number of subsequences of arr of which the sum is equal to a given sum. Write a similar recursive function subProduct that computes the number of subsequences with a given product.

5.6.5 Number of subsequences with a given length

Write a function that computes the number of subsequences with length m from an array with a given length n.

5.6.6 Print all subsequences with a given length

Write a function that prints on the screen all subsequences with length m from an array with a given length n.

5.6.7 Longest increasing contiguous subsequence

Given is the declaration of the array a:

```
int a[100];
```

You may assume that the array a is already initialised.
A contiguous subsequence is called *increasing* if each element (except for the first) is greater than its predecessor.
(a) Write a recursive function that computes the length of the longest increasing subsequence of an array.
(b) Write a recursive function that prints all longest increasing subsequences of an array on the screen.

5.6.8 Number of prime factors

Write a recursive function numPrimeFactors(**int** n) that returns the number of prime factors of the positive integer n. For example, numPrimeFactors(24) should return 4, since 24==2*2*2*3.

5.6.9 Number of uniqe prime factors

Write a recursive function numUniquePrimeFactors(**int** n) that returns the number of unique prime factors of the positive integer n. For example, numUniquePrimeFactors(24) should return 2, since there are only two unique prime factors: 2 and 3.

5.6.10 Change

We have the availability of 8 types of coins: 1, 2, 5, 10, 20, 50, 100, and 200 cents. (a) Write a recursive function that computes the *smallest* number of coins necessary to pay a certain amount using only these coins.
(b) Write a recursive function that computes the *smallest* number of coins necessary to pay a certain amount using only these coins, taking into account the possibility that you can pay more and get change in return. For example, 385 cents can be paid with two coins of 200 cents and one coin of 5 cents, and the change will be one coin of 20 cents.
(c) Modify your solution from (b) such that it prints an overview of the transaction: which coins do you pay and which coins do you get back?

5.6.11 Random walk

As you know, the city map of Manhattan is a grid with integer coordinates for the crossings of streets and avenues. The distance between crossings is measured in blocks.

We have a look at the walk of a drunk in Manhattan. At each crossing, the drunk can move in four directions: forward, backward, left and right. The aim is to determine how many 'walks' with a given number of blocks are possible from a given starting point that reach a given end point.

Write a recursive function that computes the number of possible random walks from the coordinate (x0,y0) to (x1,y1) consisting of n blocks.

5.6.12 Iterative 'tower of Hanoi'

Write a non-recursive function **void** hanoi(**int** numberOfDiscs, **int** src, **int** dest) that solves the 'tower of Hanoi' problem (see section 5.3).
Hint: Since you cannot use the recursion stack, you need to implement the stack yourself.

Chapter 6

Time complexity analysis: searching and sorting

6.1 Complexity of algorithms

To measure the efficiency of an algorithm we can look at different aspects, such as speed and memory usage. In this chapter, we will mainly be concerned with speed, although much of the material is also applicable to the analysis of memory usage.

The speed of an algorithm is typically dependent on the size of the input. In fact, we can say that the runtime of an algorithm is a function of the size of the input. The discipline that deals with the analysis of the relationship between speed and the size of the input is called *complexity theory*.

As a concrete example, consider an algorithm that sums the elements of an array that has length n. A typical implementation would be some loop over the elements of the array where the body of the loop consists of updating a sum. If we regard the body of the loop as an *elementary operation*, then we could say that the number of elementary operations that the algorithm executes is about n.

It is a bit arbitrary what we call an elementary operation. It can be the increment of a counter, but also the swapping of two elements in an array. The only restriction that we impose on the choice of an elementary operation is that it should take a fixed amount of time to execute, i.e. it should not depend on the (size of) the input.

For example, consider the following program fragment, which for a positive integer m determines the value of the expression $\sum_{i=1}^{m}(\sum_{j=1}^{i} j)$.

```
sum = 0;
for (i = 1; i <= m; i++) {
    for (j = 1; j <= i; j++) {
        sum += j;
    }
}
printf ("%d\n", sum);
```

The runtime of this code fragment is clearly dependent on the value of m. Therefore, we choose this number as a measure of the size of the 'input'. As a basic operation we choose the body of the inner loop (i.e. sum += j) of which the runtime is clearly independent of m.

For each value of i, the inner loop takes exactly i iterations, since j runs from 1 to i. In other words, the total number of elementary steps is $\sum_{i=1}^{m} i$. We now use the convenient identity (try to prove it yourself):

$$\sum_{i=1}^{m} i = m(m+1)/2$$

So, the number of elementary operations that the code fragment performs is m∗(m +1)/2. For large values of m this is roughly equal to the square of m divided by 2. Therefore, we say that the algorithm has *quadratic time complexity*.

But we can do better. Using the above identity, we can compute the same result using the following code fragment:

```
sum = 0;
for(i = 1; i <= m; i++) {
    sum += i*(i+1)/2;
}
printf ("%d\n", sum);
```

We replaced the inner loop of the previous code fragment by an assignment of which the runtime is constant (i.e. it is independent of the value m). It is clear that the algorithm now takes only m elementary steps. This algorithm computes the same result as the previous algorithm, but it has *linear time complexity*.

But we can still do better. We will make use of a second identity:

$$\sum_{i=1}^{m} i^2 = m(m+1)(2m+1)/6.$$

With some calculus we find:

$$\begin{aligned}\sum_{i=1}^{m}\left(\sum_{j=1}^{i} j\right) &= \sum_{i=1}^{m}\frac{i(i+1)}{2} = \sum_{i=1}^{m}\frac{i^2+i}{2} = \frac{1}{2}\left(\sum_{i=1}^{m} i^2 + \sum_{i=1}^{m} i\right)\\ &= \frac{1}{2}\left(\frac{m(m+1)(2m+1)}{6} + \frac{m(m+1)}{2}\right) = \frac{m(m+1)(m+2)}{6}.\end{aligned}$$

This means that we can compute the same result with a single assignment:

```
sum = m*(m+1)*(m+2)/6;
printf ("%d\n", sum);
```

The runtime of the assignment is independent of the value of m, even though m is part of the expression on the right hand side of the assignment. We say that the 'algorithm' has *constant time complexity*.

6.1.1 Big $\mathcal{O}$-notation

The exact determination of the number of elementary steps that a program makes can be very complex and is in many cases not necessary. In order to compare the complexity of algorithms we use the so-called 'big $\mathcal{O}$-notation'.

Let $f(n)$ and $g(n)$ be two functions. We say that $f(n) \in \mathcal{O}(g(n))$ if and only if there exist constants c and N such that $f(n) \leq c \cdot g(n)$ for every $n > N$. The notation $f(n) \in \mathcal{O}(g(n))$ is read as '$f(n)$ is in big $\mathcal{O}$ of $g(n)$'.

With this definition, we ensure that less interesting factors are ignored in the time complexity analysis of an algorithm and that we are left with only the essentials. The idea is that the function $g(n)$ for sufficiently large values of n (i.e. $n > N$) is always greater than $f(n)$, apart from a constant factor c. This allows us to bound $f(n)$ by a simpler function $g(n)$. Here are some examples:

- $7n + 18 \in \mathcal{O}(n)$
- $n^3 + 4n^2 \in \mathcal{O}(n^3)$
- $(2n + 5)^2 \in \mathcal{O}(n^2)$
- $n^k \in \mathcal{O}(n^m)$, for each $m \geq k$

Check yourself that each of these examples meets the definition of $f(n) \in \mathcal{O}(g(n))$ by determining suitable values for c and N. Also try to prove that $2^n \notin \mathcal{O}(n^k)$ for any k.

Using the big $\mathcal{O}$-notation, we can now compare the time complexity of algorithms without computing the number of elementary steps exactly. An algorithm that performs $(m+1)^2$ elementary operations has the same order of complexity as an algorithm that performs $3m^2 + 25m + 4$ elementary operations: they are both in $\mathcal{O}(m^2)$. It is to be expected that the runtime of the two algorithms differs only by a constant factor for sufficiently large inputs.

We ignore this constant factor in determining the order of the time complexity, but in practice this constant factor can be of real importance. For example, a movie player that can play video frames at 12 frames per second would not be acceptable, while a player that is twice as fast is perfectly fine.

6.2 Searching in an array

6.2.1 Linear search

The simplest algorithm to find a specific object in a possibly unsorted array is the *linear search* algorithm. We simply start at the first element of the array and iterate through all elements one by one until we find the element we are looking for, or until we reach the end of the array.

The function linearSearch in the following code fragment implements this algorithm. The function has three parameters: the length of the array, the array itself and the value that we are searching for. The function returns −1 in the case that the element that we are searching for does not occur in the array. Otherwise, it returns the smallest index i for which arr[i] == value.

```
int linearSearch(int length, int arr[], int value) {
  int i = 0;
  while ((i < length) && (arr[i] != value)) {
    i++;
  }
  return (i == length ? -1 : i);
}
```

In the worst case, we have to iterate over all elements of the array, i.e. n iterations. So, the time complexity of linear search in an array that has size n is in $\mathcal{O}(n)$.

The above algorithm for linear search can be optimized a little bit, although it will not change its time complexity: we say that we optimize by lowering the *constant factor*. If we are sure that the value that we are searching for occurs in the array, then we will find it before the index i runs out of bounds. In other words, we can omit the test i <length in the guard of the loop, which reduces the number of tests to be performed by a factor of two (since we only need to test arr[i] != value). This can be guaranteed temporarily by replacing the last element of the array by the value that we are searching for. We call such an array element a *sentinel*. Once the loop terminates, we replace the sentinel by the original last element of the array and decide whether the search was successful or not. The optimized linear search algorithm is given in the following code fragment:

```
int linearSearch(int length, int arr[], int value) {
  int h, i;
  if (length == 0) {
    /* empty array */
    return -1;
  }
  h = arr[length-1]; /* save a copy of last element */
  arr[length-1] = value; /* sentinel */
  i = 0;
  while (arr[i] != value) {
    i++;
  }
  arr[length−1] = h; /* restore last element */
  return (arr[i] == value ? i : −1);
}
```

Optimizations like these that lower the constant factor (which we ignore in time complexity analysis), make the code usually more complicated. This optimized version shows on a standard PC (for arrays of sufficiently large size) a performance improvement of about 20%. Some programmers will argue that this modest speed improvement does not justify more complex code. Their main argument is that (more) complicated code has a bigger chance of containing bugs. They certainly have a good point. On the other side of the spectrum are programmers that want to get the maximum performance out of their programs. It is up to you to decide where you draw the line yourself. Still, in general, it is good advice to avoid such optimizations and focus first on the correctness of the program. You can always introduce such optimizations later. In particular, I would like to quote some famous computing scientists on this matter.

- "The First Rule of Program Optimization: Don't do it. The Second Rule of Program Optimization (for experts only!): Don't do it yet." – M. A. Jackson
- "More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason – including blind stupidity." – W.A. Wulf
- "Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you have proven that's where the bottleneck is." – R. Pike
- "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified" – D. Knuth

6.2.2 Binary search

For small arrays, linear search works fine. For large arrays it costs in general (too) much computing time. If we know that the array is sorted, then we can do much better. You can compare this with searching for a word in a dictionary: we can search efficiently in a dictionary because the words are sorted alphabetically.

When we search in a sorted list, we can make considerable time savings by using a so-called *divide-and-conquer* method. A divide and conquer algorithm works by recursively breaking down a problem into subproblems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the subproblems are then combined to give a solution to the original problem.

The *binary search* algorithm is a famous divide-and-conquer algorithm for searching in a sorted array. It iteratively halves the search area (range of indices in the array) until it contains only one element. If this single element differs from the value that we were searching for, then the search was unsuccessful, otherwise it was successful.

The principle of binary search is very simple, but the implementation details are subtle. We start with a sorted array arr with size length. The array is sorted in ascending order, i.e. arr[0]$\leq$ arr[1]$\leq \ldots \leq$ arr[length−1]. Again, we search for a given value.

The first step is to compare value with the element in the middle of the array, i.e. arr[length/2]. If value is less than this element then (using the fact that the array is sorted) the problem reduces to a smaller version of the original problem: we only need to search in the first half of the array. Otherwise, if value is greater or equal than the middle element then we only need to search in the second half of the array. After the first step, we are confronted with a smaller version of the original problem, so we can repeat the process. Of course, this process stops if the sub-array under consideration contains only one element: the element that we searched for (sucessful search) or some other value (unsuccesful search).

Now we have a look at this in more detail. We introduce the index range [left, right) and maintain the invariant that the value that we seek in the array cannot occur outside this index range. We repeatedly shrink the interval [left, right) until it contains exactly one element, which is the case if left==right−1. In that case, the only thing we need to do is to check whether arr[left]==value.

Now we have a closer look at the shrinking of the interval. First we compute the index of the middle element of the interval, i.e. the index mid = (left + right)/2. Note that we use integer division, so the result is truncated if left + right is odd. Next, we compare the element arr[mid] with value:

- If value < arr[mid] then we need to search to the left of the index mid. In other words, we can reduce the interval using the assignment right=mid.
- If value >= arr[mid], then we need to search in the right hand subinterval (including mid). In other words, we can reduce the interval using the assignment left=mid.

This process stops when left−right==1, so we continue as long as right−left>1. Note that from right−left>1 it follows that left<mid<right , i.e. in each iteration the interval becomes smaller, and the process eventually stops. This reasoning is subtle, and exactly the point where bad implementations of the binary search algorithm go wrong: most of the time they terminate (usually with the correct answer), but sometimes they keep running forever because the termination argument was not thought out carefully.

A recursive implementation that follows the above analysis is given in the program fragment below.

```
int recBinarySearch(int left, int right, int arr[], int value) {
  int mid;
  /* 0 <= left < right */
  if (left + 1 == right) {
    return (arr[left] == value ? left : −1);
  }
  /* 0 <= left + 1 < right */
  mid = (left + right)/2;
  /* right − left > 1 implies left < mid < right */
  if (value < arr[mid]) {
    right = mid; /* right is lowered */
  } else {
    left = mid; /* left is raised */
  }
  return recBinarySearch(left, right, arr, value);
}

int binarySearch(int length, int arr[], int value) {
  return (length == 0 ? −1 : recBinarySearch(0, length, arr, value));
}
```

The time complexity of this algorithm is in $\mathcal{O}(\log n)$. This becomes clear when you realise that in each recursive call the length of the search interval is halved. A list of length $n = 2^k$ can be halved exactly k times, and that is exactly $\log_2(2^k) = \log_2 n$. Hence, binary search is much more efficient (for sufficiently large arrays) than linear search. For example, searching in a sorted array of about 1 million elements would take at most 21 (recursive) steps (because $2^{20} = 1048576$), while a linear search would need (on average) at least half a million iterations (in the case that the value is found): a huge improvement! In particular, if we double the size of the array, the number of computation steps doubles as well if we use the linear search algorithm, while the binary search algorithm performs only one extra step!

If you prefer an iterative (i.e. non-recursive) implementation of binary search, then you are in luck. It is trivial to convert this function into an iterative one due to a special structural property of the function recBinarySearch. The only recursive call it makes is in the last line of its body. Recursion of this type is called *tail-recursion*. This type of recursion is special since the recursive call can in fact be implemented by a jump (also called a GOTO) to the beginning of the body of the called function, i.e. there is no need to build the (implicit) recursion stack. Optimizing compilers will in fact translate tail recursive source code into non-recursive machine code with jumps. We can implement these jumps easily ourselves using a (while-)loop as you can see in the following code fragment, although it is questionable whether this code would be faster than the recursive one (because the compiler will probably perform a similar code conversion):

```
int binarySearch(int length, int arr[], int value) {
  int left = 0;
  int right = length;
  int mid;
  while (left + 1 < right) {
    mid = (left + right)/2;
    if (value < arr[mid]) {
      right = mid; /* right is lowered */
    } else {
      left = mid; /* left is raised */
    }
  }
  return (((length == 0) || (arr[left] != value)) ? −1 : left);
}
```

6.3 Sorting arrays

The efficiency of binary search is one of the main reasons why sorting algorithms are so important. There exists a wide variety of sorting algorithms that sort an array. Each algorithm has different characteristics, both in terms of time and space/memory complexity. The simplest algorithms often have time complexity in $\mathcal{O}(n^2)$: examples of these are *selection sort*, *insertion sort* and *bubble sort*.

More advance algorithms have time complexity in $\mathcal{O}(n \log n)$: examples are *merge sort* and *heap sort*. Most of these advanced algorithms are recursive. The most famous sorting algorithm is probably *quicksort*: it is also a recursive algorithm that has a worst-case complexity of $\mathcal{O}(n^2)$, but in practice it has an average time complexity that resembles the complexity of merge sort. In fact, quicksort often performs better than merge sort. The big advantage of quicksort is that it is a so-called *in-place* algorithm: its input is overwritten by the output as the algorithm executes. So quicksort does not use an extra array during the computation, while merge sort needs an auxiliary array.

All the above mentioned algorithms, except for heap sort, are discussed in the remainder of this section.

6.3.1 Selection sort

One of the simplest sorting algorithms is *selection sort*. We want to sort an array arr with size length in ascending order, i.e. we want arr[0] $\leq$ arr[1] $\leq \ldots \leq$ arr[length−1].

We introduce an index i and assume that for all indices less than i the array is already sorted. This is easily initialized with i = 0, and it is also easily kept invariant as follows. For a certain

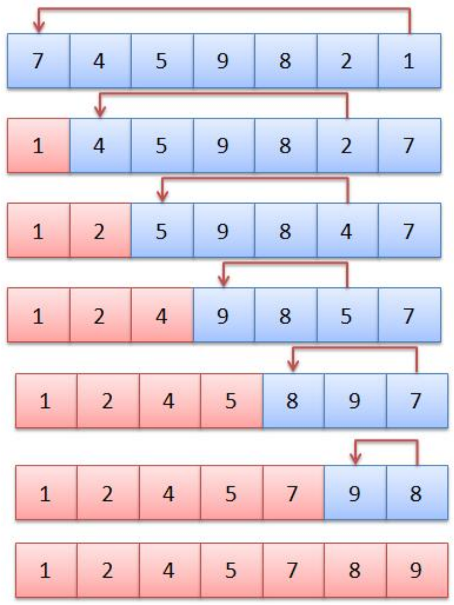


Figure 6.1: Selection sort

index i we are looking for the smallest element arr[j] where j is an index from the range [i,length). Let minimum be the index of that minimum element. Then we swap arr[i] and arr[minimum]. Note that minimum and i can be the same index, but this does not pose a problem. After this swapping operation, we can increment i, and the invariant is restored. Of course, this process is repeated until i==length (see Fig. 6.1).

The correctness of the algorithm is easy to see. This simplicity however, comes at a price. The algorithm consists of two nested loops. For each index i, a new loop over (approximately) length−i indices is performed. In total, the algorithm performs about length∗length/2 steps. Therefore, the algorithm is in $\mathcal{O}(n^2)$ where n is the length of the array. This means that, using selection sort, the sorting of an array with a thousand elements takes about 100 times longer than sorting an array with a hundred elements. Consequently, selection sort is hardly ever used for sorting large arrays. The algorithm is shown in the following code fragment.

```
void swap(int i, int j, int arr[]) {
  int h = arr[i];
  arr[i] = arr[j];
  arr[j] = h;
}

void selectionSort(int length, int arr[]) {
  int i, j, minimum;
  for (i=0; i < length; i++){
    minimum = i;
    for (j=i+1; j < length; j++) {
      if (arr[j] < rij[minimum]) {
        minimum = j;
      }
    }
    swap(i, minimum, arr);
  }
}
```

6.3.2 Insertion sort

Another simple sorting algorithm is *insertion sort*. It iterates with an index i over the array, starting from index 1. In each iteration the value a[i] is inserted in the already sorted sequence a[0],..,a[i−1]. This process is repeated until i==length. It is is depicted graphically in Fig. 6.2.

We now have a look at the implementation details of this idea. We start by introducing the index i and keep invariant that the subarray to the left of i is already sorted. This is easily initialized if we start with the index i=1, because the subarray consisting of only the element a[0] is automatically sorted. We can maintain the invariant as follows. In each iteration, we search the location idx in the left subarray where we must insert the value a[i]. We use the following function to find idx. It is a variant of the linear search algorithm.

```
int searchIndex(int length, int arr[], int value) {
  int i = 0;
  while ((i < length) && (arr[i] <= value)) {
    i++;
  }
  return i;
}
```

With the help of the function searchIndex, it is easy to find the location where we need to insert arr[i] using the assignment idx =searchIndex(i, arr, arr[i]). Using this call, the function searchIndex will return the smallest index j (with 0<=j<idx), such that arr[i] > value or i in case such a value does not exist in the subarray. Now that we know the location idx, we need to shift all elements between idx and i−1 one position to the right, in order to make space for the insertion of a[i]. We use the function shiftRight for this.
The complete code of the insertion sort algorithm is given below.

```
int searchIndex(int length, int arr[], int value) {
  int i = 0;
  while ((i < length) && (arr[i] <= value)) {
    i++;
  }
  return i;
}
void shiftRight(int length, int arr[], int idx) {
  /* shift the elements with indices from [idx,length) one position to the right. */
  int i;
  for (i=length; i > idx; i−−) {
    arr[i] = arr[i−1];
  }
}

void insert(int length, int arr[], int value) {
  /* insert value in sorted array */
  int idx = searchIndex(length, arr, value);
  shiftRight(length, arr, idx);
  arr[idx] = value;
}

void insertionSort(int length, int arrj[]) {
  int i;
  for (i=1; i < length; i++) {
    insert(i, arr, arr[i]);
  }
}
```

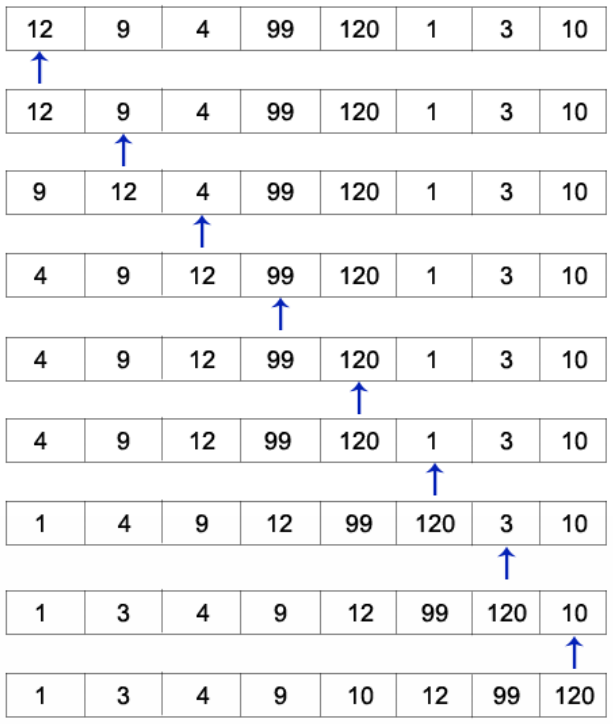


Figure 6.2: Insertion sort

The time complexity analysis of insertion sort and selection sort is similar. Both algorithms are in $\mathcal{O}(n^2)$ where n is the length of the array. After all, the routine insertionSort contains a loop running from 1 to length. In each iteration we call the function insert which in turn calls the functions searchIndex and shiftRight. The latter two functions both contain a loop over (some of) the elements, so we are in fact dealing with nested loops.

Note that the time complexity is not improved by replacing the linear search algorithm in searchIndex by a binary search. After all, we must still use the function shiftRight which has a linear time complexity, so it will dominate the logarithmic time complexity of the binary search algorithm. More formally, $\mathcal{O}(n(\log n + n)) = \mathcal{O}(n^2)$.

Although both algorithms are in $\mathcal{O}(n^2)$, their runtimes will in general not be the same. For example, consider the (trivial) case in which the input of the algorithm is a sorted array. For the selection sort algorithm this makes no difference, it always performs about length*length/2 steps. On the other hand, the insertion sort algorithm will perform only a single loop over the input, and will discover that it does not need to do anything at all, i.e. it performs only length steps. We conclude that in general insertion sort runs faster than selection sort, although both algorithms have the same time complexity: the constant factor of insertion sort is significantly smaller than the constant factor of selection sort.

6.3.3 Bubble sort

The third sorting algorithm that we will study is *bubble sort*. The algorithm works as follows. With an index j we iterate over the elements of the array and compare each element with its neighbouring element (to the right). If row[j] > row[j+1], then we swap the elements. After this operation is performed for all indices j, the largest element will be arr[length−1]. We cannot say much about the other elements of the array. After this first pass over the array, we can repeat this

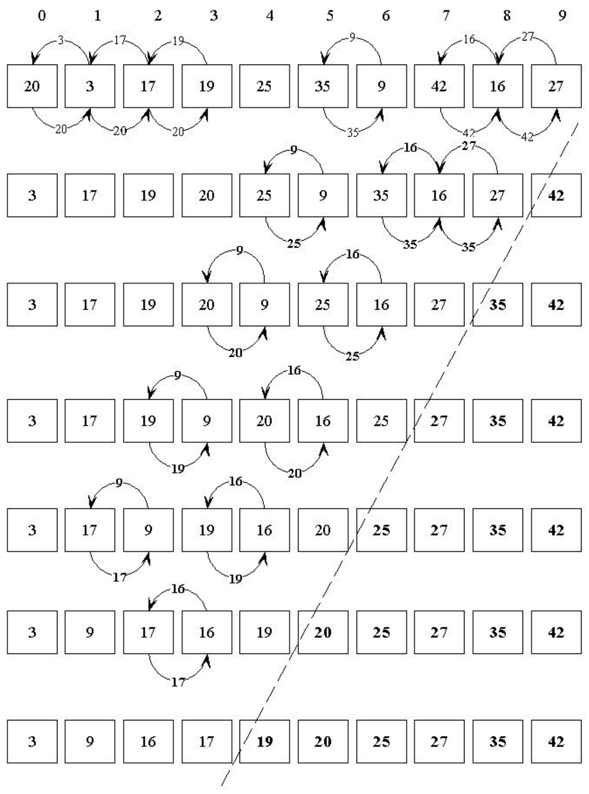


Figure 6.3: Bubble sort: values below the dotted line are already in sorted order.

process, however this time we will not visit the last element (because we already know that it is at the right place). It is not difficult to see that after performing length passes, the entire array is sorted.

Note that in pass i, it is sufficient to process the indices j from the interval [0,length−i), because the last i−1 elements are already in place. This is depicted graphically in Fig. 6.3 by a dotted line.

During execution of this algorithm we see the larger elements slowly 'float' to the top of the array, like bubbles in water. This sorting algorithm derives its name from this metaphor. The algorithm is given in the following code fragment.

```
void bubbleSort(int length, int arr[]) {
  int i, j;
  for (i=0; i < length; i++) {
    for (j=0; j+1 < length−i; j++) {
      if (arr[j] > arr[j+1]) {
        swap(j, j+1, arr);
      }
    }
  }
}
```

Because of the nested loops, it is clear that bubble sort is also in $\mathcal{O}(n^2)$ where n is the length of the array. By a simple optimisation we may reduce the running time a bit (albeit it does not change the time complexity of the algorithm). If the algorithm detects that in some pass no swaps were performed, then the array is already sorted. Hence, no further iterations are needed, and the algorithm terminates. The following implementation of bubble sort makes use of an extra variable `change` (which acts as a Boolean variable) to makes use of this observation.

```
void bubbleSort(int length, int arr[]) {
  int i, j, change;
  for (i=0; i < length; i++) {
    change = 0;
    for (j=0; j+1 < length−i; j++) {
      if (arr[j] > arr[j+1]) {
        swap(j, j+1, arr);
        change = 1;
      }
    }
    if (change == 0) {
      /* nothing changed in this pass, so we can stop */
      break;
    }
  }
}
```

6.3.4 Merge sort

Merge sort is a recursive sorting algorithm, based upon the divide-and-conquer principle. Merge sort works by splitting the array to be sorted into two subarrays of (almost) equal size. These two unsorted arrays have a smaller length than the original list, so we can recursively sort both sublists. The base cases of this recursion are subarrays that contain at most one element, hence these are automatically sorted. So, in the base case the function returns immediately. Upon return from the recursion, the sorted subarrays are merged together. The divide-and-conquer structure of the algorithm is depicted graphically in Fig. 6.4.

The merging phase has linear time complexity. It works similarly to the merging of two piles of playing cards. Each pile is sorted and placed face-up on a table with the smallest cards on top. We will merge these into a single sorted pile, face-down on the table by iterating the following procedure. Choose the smaller of the two top cards. Remove it from its pile, thereby exposing a new top card. Place the chosen card face-down onto the output pile. This process is iterated until one of the piles is empty. In that case we take the remaining pile and place it face-down onto the output pile.
The complete merge sort algorithm is given in the following code fragment. Please study the details yourself. Especially the merging phase is instructive.

```
int *copySubArray(int left, int right, int arr[]) {
  int i;
  int *copy = dynamicIntArray((right − left)*sizeof(int));
  for (i=left; i < right; i++) {
    copy[i − left] = arr[i];
  }
  return copy;
}

void mergeSort(int length, int arr[]) {
  int l, r, mid, idx, *left, *right;
  if (length <= 1) { /* base case */
    return;
  }
  /* recursive case: length >= 2 implies 0 < length/2 < length */
  mid = length/2;
  /* make copies of subarrays and sort them (recursion) */
  left = copySubArray(0, mid, arr);
  right = copySubArray(mid, length, arr);
  mergeSort(mid, left);
  mergeSort(length − mid, right);
  /* merge sorted subarrays */
  idx = 0;
  l = r = 0;
  while ((l < mid) && (r < length − mid)) {
    if (left[l] < right[r]) {
      arr[idx] = left[l];
      l++;
    } else {
      arr[idx] = right[r];
      r++;
    }
    idx++;
  }
  /* process trailing elements: either l < mid or r < length − mid */
  while (l < mid) {
    arr[idx] = left[l];
    idx++;
    l++;
  }
  while (r < length − mid) {
    arr[idx] = right[r];
    idx++;
    r++;
  }
  /* deallocate memory */
  free(left);
  free(right);
}
```

The question arises whether merge sort performs better than the algorithms from $\mathcal{O}(n^2)$ that we have studied previously. This is indeed the case, because merge sort is in $\mathcal{O}(n \log n)$. For an arbitrary array of length n this may be difficult to see directly, but in case $n = 2^k$ for some natural number k it is much easier to see. We therefore focus on this case.

We try to derive an expression $T(n)$ to determine the number of basic steps that merge sort makes in sorting an array of length n. We choose as a basic step the number of comparisons between elements of the array. In the above fragment this is the number of times that the comparison left [l] < right[r] is performed in the merge phase. The number of comparisons that the merge phase makes for two subarrays that together have length n is clearly at most $n-1$. Due to the recursion however, these subarrays themselves were constructed at a deeper recursion level: both took about $T(n/2)$ steps. This analysis leads to the following recurrence relation:

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n - 1 & \text{if } n = 2^k \text{ for } k > 0 \end{cases}$$

The solution of this recurrence is

$$T(n) = n \log_2 n - n + 1.$$

This is easy to prove using induction. We start with $T(1) = 0 = \log_2 1 = 1 \cdot \log_2 1 - 1 + 1$. So, the base is true. Now we must show that $T(2n) = 2n \log_2(2n) - 2n + 1$ under the assumption

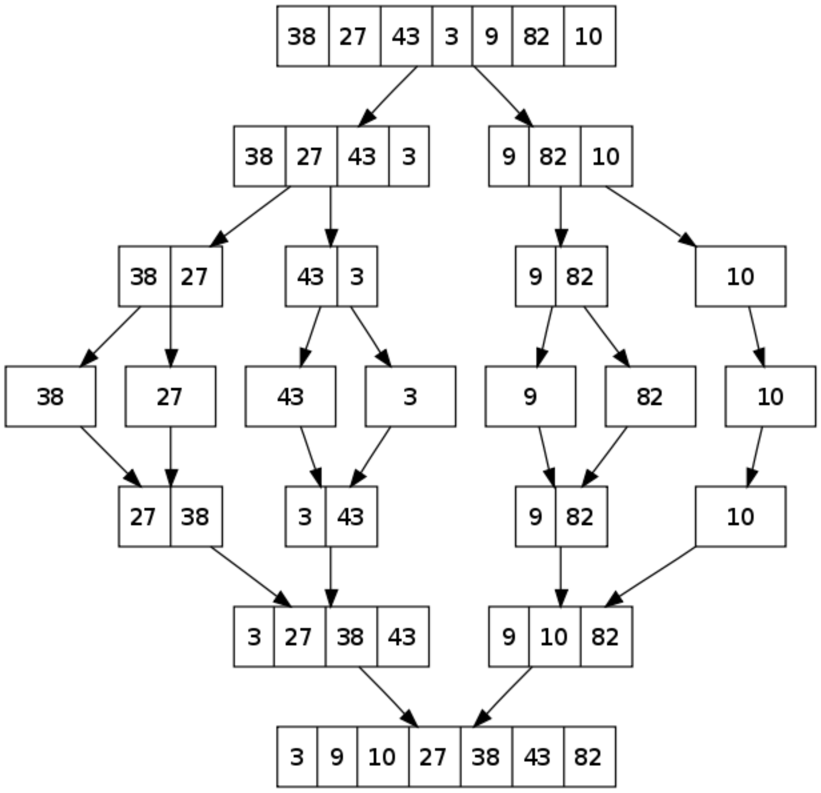


Figure 6.4: Merge sort: a divide-and-conquer sorting strategy.

(induction hypothesis) that $T(n) = n\log_2 n - n + 1$:

$$\begin{aligned} T(2n) &= 2T(n) + 2n - 1 = 2(n\log_2 n - n + 1) + 2n - 1 = 2n\log_2 n + 2n - 2n + 1 \\ &= 2n\log_2 n + 2n\log_2 2 - 2n + 1 = 2n(\log_2 n + \log_2 2) - 2n + 1 \\ &= 2n\log_2(2n) - 2n + 1 \end{aligned}$$

Using mathematical induction we showed that $T(n) = n\log_2 n - n + 1$, hence merge sort is in $\mathcal{O}(n\log n)$ where n is the length of the array to be sorted. So, merge sort is a much faster algorithm than all the previously discussed sorting algorithms that were all in $\mathcal{O}(n^2)$. This speed, however, came at a (small) price: we need auxiliary memory (while the other algorithms were in-place algorithms). Another famous sorting algorithm, called *heap sort*, is also in $\mathcal{O}(n\log n)$ while it is also in-place. We will not discuss this algorithm in this course, but it will be discussed in the course '*Algorithms and Datastructures*'.

6.3.5 Is $\mathcal{O}(n\log n)$ optimal?

In section 6.3.4 we concluded that merge sort is in $\mathcal{O}(n\log n)$. Now we ask ourselves the question whether we can do even better than $\mathcal{O}(n\log n)$.

Unfortunately, the answer to this question is NO. It is not possible to design a better sorting algorithm *which is based on comparison of elements of the array.* This claim may sound strange, after all how can you make a claim about all sorting algorithms without knowing their implementations? This section briefly discusses this issue.

We consider sorting algorithms based on comparisons of elements of the array. We do not make any assumption about the array itself, except that its length is some natural number n.

A sorted array that is obtained from an unsorted array is nothing else than a specific permutation of the original array. A sequence is a *permutation* of some other sequence if it contains the same elements as the original one in (possibly) another order. There exist $n!$ (factorial of n) permutations of a sequence of n elements (provided that the elements are all different, we ignore this technical detail in the rest of this discussion). A good sorting algorithm would yield the same sorted output array given any of its permutations as its input.

Of course, the result of the comparison `arr[i]` < `arr[j]`, for some indices `i` and `j`, is binary, i.e. it can only yield two results: true or false. The question now arises how many binary comparisons you need at least to select a specific permutation from all these $n!$ permutations. For each permutation, the sorting algorithm follows a different computation order in the *decision tree* of the algorithm which is based on binary comparisons. Any comparison halves the number of remaining possibilities. Let C be the number of comparisons that we have to make. Apparently, it must be true that $n! \leq 2^C$, because we must be able to find every possible permutation using C binary comparisons. We can solve this inequality by taking the logarithm on both sides. So, we find $C \geq \log(n!)$.
We can bound the expression $\log(n!)$:

$$\frac{n \log n}{2} - \frac{n}{2} \leq \log(n!) \leq n \log n.$$

The inequality $\log(n!) \leq n \log n$ is easy to prove if we use the identity $\log(a \cdot b) = \log a + \log b$:

$$\log(n!) = \log\left(\prod_{i=1}^{n} i\right) = \sum_{i=1}^{n} \log i \leq \sum_{i=1}^{n} \log n = n \log n.$$

The proof that $(n \log n)/2 - n/2 \leq \log(n!)$ requires somewhat more creativity:

$$\log(n!) = \sum_{i=1}^{n} \log i \geq \sum_{i=n/2}^{n} \log i \geq \sum_{i=n/2}^{n} \log(n/2) = \sum_{i=n/2}^{n} (\log n - \log 2) = \frac{n \log n}{2} - \frac{n}{2}.$$

Now that we have shown that a lower bound for C is of the order $n \log n$, the inevitable conclusion is that it is not possible to make a new sorting algorithm (that is based solely on comparison) that has a better time complexity than any of the other sorting algorithms in $\mathcal{O}(n \log n)$.

6.3.6 Quicksort

Quicksort is another recursive sorting algorithm that is based on the divide-and-conquer principle. Just as in the case of merge sort, the key idea is to split the array to be sorted in subarrays, and sort these subarrays recursively. The main difference lies in the partitioning of the array in subarrays. We will see that this partitioning strategy of quicksort is in-place, i.e. it does not require auxiliary memory.

The quicksort algorithm works as follows. Some (arbitrary) element from the array to be sorted is chosen. This element is called the *pivot*. After the pivot is chosen, the array is permuted in such a way that it consists of three consecutive regions. All elements in the first region have a value that is less than or equal to the pivot. The second 'region' is a singleton. It contains only the pivot itself. The third region contains all the elements that have a value greater than the pivot (see Fig. 6.5).

The function that performs this partitioning is traditionally called `partition`. The function plays two roles: it permutes the array and it returns the position of the pivot in the permuted array. Many different implementations of this function have been proposed in the literature. They all have the same time complexity: `partition` is in $\mathcal{O}(n)$.

After the rearrangement of the elements by `partition`, we need to sort the first and the third region recursively. The middle region (containing only the pivot) is already placed at the right position in the array. The code is given in the following code snippet. Study it carefully, especially the actual parameters of the second recursive call are a bit tricky.

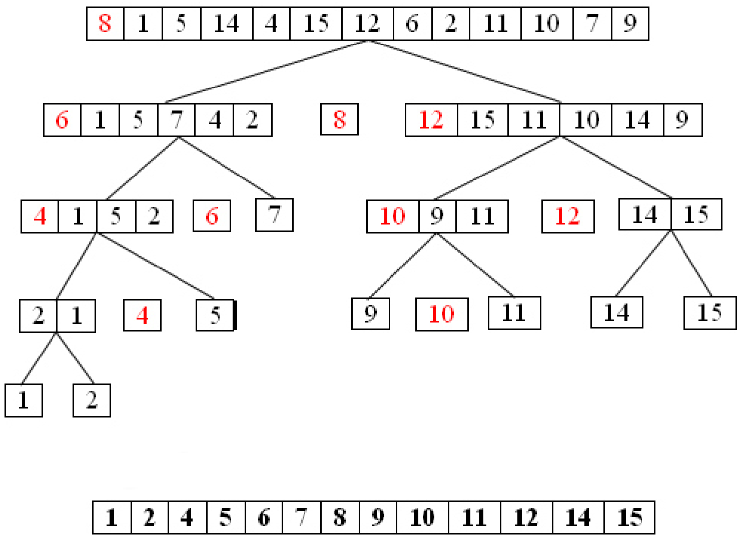


Figure 6.5: Quicksort: a divide-and-conquer sorting strategy

```
void quickSort(int length, int arr[]) {
  int boundary;
  if (length <= 1) {
    return; /* empty or singleton array: nothing to sort */
  }
  boundary = partition(length, arr);
  quickSort(boundary, arr);
  quickSort(length − boundary − 1, &arr[boundary + 1]);
}
```

We now concentrate on the implementation of the function partition. The function permutes the arrays as described above and returns the position of the pivot. Let boundary be the final position of the pivot in the array. We strive to establish that arr[i]≤arr[boundary] for 0≤i<boundary and arr[boundary]<arr[i] for boundary<i<length.

For the pivot we can choose any arbitrary element of the array. In this presentation we always choose the first element of the array as the pivot, i.e. pivot=arr[0]. Next, we introduce the indices left and right, and maintain the following invariant:

arr[i]≤pivot (for all i in [0,left)) and pivot<arr[j] (for all j in [right,length))

We loop with an index i over the array elements (i.e. i ranges from 0 to length−1), and maintain the following invariant: arr[j]≤pivot (for 0≤j<boundary) and arr[j]>pivot (for boundary≤j<i). We can initialise this trivially using the following two assignments:

```
  int left = 0;
  int right = length;
```

Now we want to increment left and decrement right, while maintaining the invariant. We stop, of course, when we reach the situation with left>=right. It is clear that we can increment left under the condition (left < right)&& (arr[left] <= pivot):

```
    while ((left < right) && (arr[left] <= pivot)) {
      left++;
    }
```

The situation is symmetric for the decrement of right:

```
  while ((left < right) && (pivot < arr[right−1])) {
    right−−;
  }
```

At the end of these two loops, we are in the situation that either left>=right or (arr[left] > pivot) && (arr[right−1] <= pivot). In the first case, we are done. In the latter case, we swap the elements arr[left] and arr[right−1] and go on.

At the end of this process, we are confronted with two regions: the region to the left of the index left (so without the index left itself) with elements that are less or equal to the pivot, and a region that starts at index left with elements that are greater than the pivot. Note that during the execution of partition the pivot remains at arr[0]. So, what remains to be done is to swap arr[0] and arr[length−1] and return the location of the pivot, i.e. left−1.

This analysis leads to the following implementation of the function partition. Please study the details carefully.

```
int partition(int length, int arr[]) {
  int left = 0;
  int right = length;
  int pivot = arr[0];
  while (left < right) {
    while ((left < right) && (arr[left] <= pivot)) {
      left++;
    }
    while ((left < right) && (pivot < arr[right−1])) {
      right−−;
    }
    if (left < right) {
      /* (arr[left] > pivot) && (arr[right−1] <= pivot) : swap */
      right−−;
      swap(left, right, arr);
      left++;
    }
  }
  /* put pivot in right location: swap(0,left−1,arr) */
  left−−;
  arr[0] = arr[left];
  arr[left] = pivot;
  return left;
}
```

Of course, we ask ourselves the question whether quicksort is an improvement over the best sorting algorithm that we know so far, namely merge sort. In the worst case, quicksort is very slow: this is the case in which the input of the algorithm is the sorted array. In this situation, the routine partition does not work well: it always produces an empty region, a singleton region and the remaining $n-1$ elements in the third region. This bad partitioning yields a quadratic worst-case time complexity, i.e. quicksort is in $\mathcal{O}(n^2)$.

So, why is the algorithm called quicksort? To answer this, we have a look at the best case, in other words the case in which the pivot is always chosen such that partition splits the array in (almost) equal sized halves. In other words, the pivot ends always in the middle. In this special case the time complexity of quicksort is of the order $n \log n$, using a similar argument as we used for merge sort. In practice, the algorithm will be even faster than merge sort, because it does not use extra memory and copying of data (quicksort is in-place). As a result, the constant factor of quicksort is considerably smaller than the constant factor of merge sort.

Of course, the input of quicksort is not likely to be the best case. Still, it turns out that the choice of the pivot is not as critical as you might expect. For example, let us assume that we choose the pivot such that one of the subarrays produced by partition contains about 10% of the elements (an imbalance of 1:10). Using a similar analysis as in the case of merge sort we can find a recurrence relation $T(n)$ for the number of computation steps that quicksort makes on an array of length n:

$$T(n) = \begin{cases} 0 & n = 1 \\ T(9n/10) + T(n/10) + n & n > 1 \end{cases}$$

The recursion tree of this splitting has $\log_{10/9}(n) \approx \log_{1.11}(n) = \log n / \log 1.11$ levels, so we still find a complexity of the order $n \log n$. We will not give an exact proof of this claim here, although it is a good exercise to try to prove it yourself.

How can we ensure that we partition well very often? A simple solution is to not always choose the first element, but to choose a random element from the list. For this purpose we use the function rand from the standard C library. This function returns an arbitrary integer greater than or equal to 0. So, we can choose a random pivot using the assignment pivot=arr[rand()%length]. An algorithm that uses this trick is called a *randomized algorithm*. The following randomized version of partition will result in a much better performance of quicksort, even if the input is a sorted array.

```
int partition(int length, int arr[]) {
  int left = 0;
  int right = length;
  int idx, pivot;

  swap(0, rand()%length, arr); /* place a random element at index 0 */
  pivot = arr[0];

  /* the rest of the code is the same */
  while (left < right) {
    while ((left < right) && (arr[left] <= pivot)) {
      left++;
    }
    while ((left < right) && (pivot < arr[right-1])) {
      right--;
    }
    if (left < right) {
      /* (arr[left] > pivot) && (arr[right-1] <= pivot) : swap */
      right--;
      swap(left, right, arr);
      left++;
    }
  }
  /* put pivot in right location: swap(0,left-1,arr) */
  left--;
  arr[0] = arr[left];
  arr[left] = pivot;
  return left;
}
```

6.3.7 Counting sort

From the discussion in section 6.3.5 we know that it is not possible to make a sorting algorithm that, on the basis of mutual comparison of elements in an array, has a time complexity better than $\mathcal{O}(n \log n)$.

This does not mean that we cannot design a faster sorting algorithm in special cases. An example of such an algorithm is *counting sort*. It is an extremely simple sorting algorithm, which can only be used if the values in the array to be sorted are integral (i.e. not floats) and have a (very) limited range in size. A good example is a big array with integers (for example, 1000000 elements), wherein the elements are between 0 and 255 (i.e., byte-values).

Let min be the minimum value in the array, and max the maximum value. Now we can create a histogram of size 1+max−min:

```
int *count = calloc(1 + max − min, sizeof(int));
```

In this histogram we simply count for each value between min and max how many times it occurs in the array. Given the complete histogram, we can simply generate the sorted array, without any need for comparison nor swapping.

The counting sort algorithm is given in the following code fragment:

```
void countingSort(int length, int arr[]) {
  int min, max, range;
  int i, j, idx, *count;
  /* compute range of values */
  min = max = arr[0];
  for (i=1; i < length; i++) {
    min = (arr[i] < min ? arr[i] : min);
    max = (arr[i] > max ? arr[i] : max);
  }
  range = 1 + max − min;
  /* create histogram and fill it with zeroes (i.e. use calloc) */
  count = calloc(range, sizeof(int));
  /* count occurrences */
  for (i=0; i < length; i++) {
    count[arr[i] − min]++;
  }
  /* generate sorted array */
  idx = 0;
  for (i=0; i < range; i++) {
    for (j=0; j < count[i]; j++) {
      arr[idx] = min + i;
      idx++;
    }
  }
  /* release memory used for histogram */
  free(count);
}
```

This algorithm is clearly in $\mathcal{O}(n+k) = \mathcal{O}(n)$, where n is the length of the array, and k the size of the range (and $k \leq n$). In other words, counting sort has linear time complexity. The reason why this algorithm broke the $\mathcal{O}(n \log n)$ limit is twofold: the algorithm is not based on comparison of elements, and the range of the numbers is limited in such a way that we can afford to allocate a histogram of a sufficient size.

6.4 Tutorial exercises

6.4.1 Complexity

Let $f(n) = 2 \cdot n^3$, $g(n) = 19 \cdot n^2$, and $h(n) = 3 \cdot (n+3)^3$. Prove or refute:

1. $f(n)$ is $\mathcal{O}(g(n))$
2. $f(n)$ is $\mathcal{O}(h(n))$
3. $g(n)$ is $\mathcal{O}(f(n))$
4. $g(n)$ is $\mathcal{O}(h(n))$
5. $h(n)$ is $\mathcal{O}(f(n))$
6. $h(n)$ is $\mathcal{O}(g(n))$

6.4.2 Manhattan walks

We are located at the origin of a grid with integer coordinates and want to walk to (i, j) (where $i \geq 0$ and $j \geq 0$). At each grid point, we are allowed to move one step to the north or to the east.

1. Design a recursive function $F(i, j)$ that computes the number of possible walks from $(0, 0)$ to (i, j).
2. Find a closed expression (i.e. a formula without recursion) for $F(i, j)$ and prove its correctness by means of an inductive proof.

6.4.3 Square roots

The *integer root* of a natural number x is the greatest integer r such that r∗r <= x. In this exercise we will implement two different versions of the function isqrt:

```
int isqrt(int x); /* returns greatest integer r such that r*r <= x */
```

(a) Linear search: use a linear search algorithm to implement the body of the function isqrt.

(b) Binary search: use a binary search algorithm to implement the body of the function isqrt. Introduce two variables p and q and maintain the invariant p∗p <= x < q∗q.

(c) Determine the time complexity of both algorithms. Which version is more efficient?

6.4.4 Bisection method for finding roots

In this exercise, we consider the function $f(x) = a \cdot x^3 + b \cdot x^2 + c \cdot x + d$. The values a, b, c and d are natural numbers.

Write a function that, using binary search, returns a **double** value x such that the absolute value of $f(x)$ is less than 0.0001.

6.4.5 A two-dimensional landscape

Consider the following declaration of the two-dimensional array alt:

```
int alt[N][N];
```

One can think of alt as a landscape, where alt[x][y] denotes the altitude at location (x,y).

(a) [Easy] Write a code fragment that counts the number of points that have an altitude below a given altitude w. The time complexity of your algorithm should be quadratic in N. For example, if w==20, we count the number of **bold face** figures in the following matrix:

1	**16**	25	22	**0**	**1**	**17**	20	**19**	29
9	22	**7**	**1**	**5**	**16**	**13**	**3**	**14**	24
12	**6**	**13**	**16**	**14**	20	**9**	**14**	**11**	**6**
16	**0**	**2**	**13**	**8**	**2**	**16**	**14**	**3**	**16**
25	**16**	20	27	**7**	**3**	**5**	27	24	22
23	23	**2**	29	**14**	26	26	**14**	**8**	**19**
25	**19**	**9**	**18**	29	20	27	**15**	**8**	**18**
27	20	27	**12**	21	**1**	**14**	**12**	**6**	26
16	**7**	**8**	**12**	**3**	**16**	**15**	**15**	**18**	**0**
13	**2**	**11**	29	**9**	23	**15**	24	**7**	**12**

(b) [Intermediate] Again, we consider the array arr, but this time there are some extra restrictions on it:

- If x0 <= x1 then alt[x0][y] <= alt[x1][y] for all y.
- If y0 <= y1 then alt[x][y0] <= alt[x][y0] for all x.

One can think of alt as a landscape of which the altitude increases (or at least does not decrease) if one moves to the east or north (or north-east).

7	**13**	**14**	25	25	27	29	29	32	33
6	**11**	**12**	23	24	25	27	29	32	32
6	**9**	**12**	22	22	23	27	29	30	30
6	**9**	**10**	20	20	23	25	25	27	28
6	**9**	**10**	**18**	20	21	21	23	25	25
6	**7**	**10**	**16**	**16**	**19**	21	22	23	23
5	**5**	**8**	**14**	**15**	**17**	**19**	21	21	23
5	**5**	**6**	**12**	**12**	**15**	**16**	**17**	**18**	**19**
5	**5**	**6**	**10**	**12**	**14**	**15**	**16**	**17**	**19**
3	**5**	**6**	**8**	**9**	**9**	**9**	**10**	**11**	**13**

Write a code fragment that, given the above restrictions on alt, counts the number of points that have an altitude below a given altitude w. The time complexity of your algorithm should be of the order N*log N. *Hint: Use a binary search per row.*

(c) [Hard] The same exercise as in (b), but now with a linear time complexity.

6.4.6 Minimum and maximum of an array

The following function returns the minimum value in a non-empty array of integers:

```
int minimum(int len, int arr[]) {
  int i, min = arr[0];
  for (i=1; i < len; i++) {
    if (arr[i] < min) {
      min = arr[i];
    }
  }
  return min;
}
```

Clearly, the function needs len−1 comparisons to compute its result.

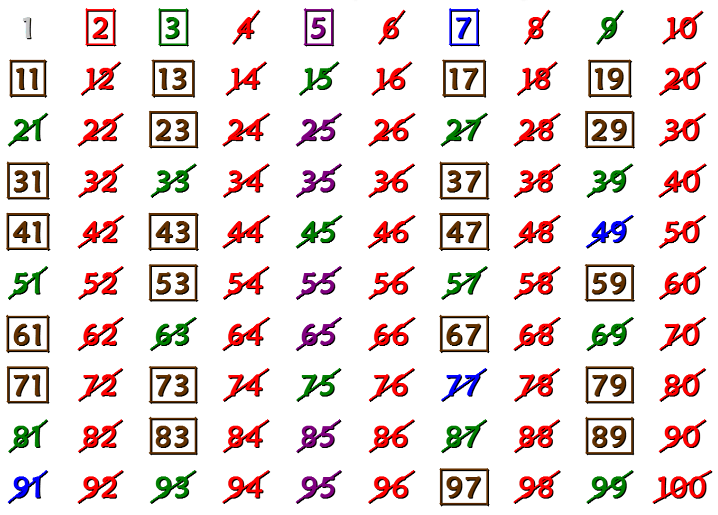
Write a function minmax that computes the minimum and the maximum value of a (non-empty) sequence with length len. The function should not need more than 3*len/2 comparisons. Use the following prototype:

```
void minmax(int len, int arr[], int *min, int *max);
```

6.4.7 Sieve of Eratosthenes

The Ancient Greeks already knew a method to compute the list of all primes smaller than a certain limit n. The same procedure is still used today, and is called the *sieve of Eratosthenes*.

The method works as follows. We start with the complete list of all the numbers from 2 to n. The first number of this list (i.e. 2) is prime, and is printed on the screen. Then we remove all multiples (including 2 istelf) from the list. Now, the first element of the remaining list is prime again: it gets printed and all its multiples are removed. This procedure is repeated until the list is empty.



(a) Write a program fragment that prints all primes less than a given n on the screen.
(b) Write a program that returns the list of all primes less than a given n.

6.4.8 Grid points on a disc

The following code fragment can be used to compute how many grid points are on a closed (i.e. the edge included) circular disc with center (0,0) and radius r>0. The radius is a positive integer.

```
  int x, y, cnt = 0;
  for (x = 1; x <= r; x++) {
    for (y = 1; y <= r; y++) {
      if (x*x + y*y <= r*r) {
        cnt++;
      }
    }
  }
```

Note that it suffices to count only the grid points that lie on a quarter of the disc. The real answer can be computed easily from this result. The above code fragment has time complexity which is quadratic in r. Write a better algorithm that computes the same result with a complexity that is linear in r.

6.4.9 Floyd-Warshall algorithm

We consider N villages: they are enumerated 0,1,..,N−1. Some of these villages are directly connected by a road, while others are not. However, from each village any other village is reachable directly or via one or more other villages.

We declare a distance matrix:

```
int dist[N][N];
```

The value dist[i][j] (which is equal to dist[j][i]) is non-zero if there exists a direct road from i to j. In that case, dist[i][j] denotes the distance between i and j. If dist[i][j]==0, then this means either i==j or there is no direct road between i and j.

Write a program fragment that computes the length of the shortest path between i and j for all pairs (i,j). The time complexity of your algorithm should be $O(N^3)$.
Hint: Consider a shortest path from i to j via k. Then the subpath from i to k is also optimal. The same holds for the subpath from k to j.

Index