# C for Embedded Systems

**Ir. Sofie Beerens**

Embedded systems

# Table of Contents

# List of illustrations

## List of figures

## List of tables

## List of code examples

# Introduction

To understand the advantages of using C as programming language for embedded systems, we need to go back in history.

The origin of C is closely tied to the development of the Unix operating system. C was developed by Dennis M. Ritchie and Brian W. Kernighan. They decided to rewrite the full Unix operating system in the B language, developed by Ken Thompson. The need to improve the B language to overcome some shortcomings, led to the development of C.

Although, the C language was brought to the market already in 1972, it only became popular in 1978 after the publication of the book "The C programming language" (written by both C language inventors).

C is a flexible, well-structured and very compact language. It was designed to provide low-level access to memory, to provide language constructs that map efficiently to machine instructions, and to require minimal run-time support. Therefore, C was useful for many applications that had formerly been coded in assembly language, such as in system programming.

Despite its low-level capabilities, the language was designed to encourage cross-platform programming. A C program can be compiled for a very wide variety of computer platforms and operating systems. The language has become available on a very wide range of platforms, from embedded microcontrollers to supercomputers.

Programming in C needs to be done very carefully. As C provides you with a lot of freedom and flexibility, it is important to be aware of what exactly you are doing. The C compiler assumes you are an experienced designer that takes care of accessing hardware the correct way. A well-known saying about C is: "C provides enough rope to hang yourself." The aim of this course is to provide you with the right handles to start writing your own (complex) programs.

Many other programming languages were influenced by C like e.g. C#, Java, JavaScript, PHP, …

**Learning outcomes**

At the end of this course students are able to:

- analyze existing C code and understand how an existing program works. Students can read and predict the outcome of programs written in C syntax.
- create a well-structured program in C code containing functions. To accomplish this, students must be able to logically analyze concrete problems, divide these problems into smaller sub problems, convert each one of those sub problems into algorithms and translate each one of those algorithms into C syntax.
- choose the appropriate data structures to represent the different variables in the C program.
- allocate memory dynamically.
- perform operations on bits using C syntax.
- create a C program using file handling.

**Assessments**

The theory will be assessed in a written exam.
Questions will mainly check the comprehension of the C syntax in all its aspects.

The practical work will be assessed in a practical exam on a computer. A C program and corresponding flowchart must be built to solve one or more problems. Assessment will be based on quality of the flowchart, outcome of the program, algorithms chosen, usage of functions with parameters, correct choice of data structures, correct memory allocation, efficiency, programming layout, readability and reusability of the C code.

# 1 Programming languages

**Objectives**

This chapter highlights the difference between computer language and different programming languages. At the end of this chapter, one should understand how a high-level language program is processed and the type of errors that can occur during the different processing steps.

## 1.1 Introduction

In today's life, computers can no longer be ignored. Many people have access to a PC and use it to perform all sorts of tasks like searching the internet for information, computing a household budget, getting in contact with friends and family, storing pictures, …

Next to this, smaller and less obvious versions of computers can be found in all sorts of embedded systems like smartphones, tablets, camera's, car control systems, heartrate monitors and more.

The reason for using computers in all these systems is that computers can perform computations and make logical decisions much faster than human beings can. To accomplish this, the computer needs to be driven by a sequence of instructions called a computer program. The set of programs enabling the usage of the computer system is referred to as software.

Programs are written in programming languages. Such a language must contain certain elements to:

- allow efficient information storage in memory
- allow communication with the user through mouse, keyboard, screen, …
- allow reading of large amounts of data from hard disk, DVD,… and writing results to these memory devices.
- use the processor to make computations.

Some of these languages contain instructions that are directly understandable by computers. Other require intermediate translation steps.

## 1.2 Machine languages (first generation)

Machine languages can be understood directly by the computer. As different processors have different hardware architectures, a computer can only directly understand its own machine language.

A program written in machine language constists of a sequence of binary processor instructions where every instruction consists of a sequence of bits.

```
10100101
01100000
01100101
01100001
10000101
01100010
```

**Figure 1: example of machine language code**

The piece of code shown above is a program for a 6502 processor that adds the content of memory location 0x60 (hexadecimal) to the content of memory location 0x61 and stores the result in memory location 0x62.

To improve readability, these bit sequences are often written in hexadecimal notation resulting for this example in:

```
A5
60
65
61
85
62
```

**Figure 2: example of machine language code written in hex notation**

As can be seen from the example above, these programs are very difficult to understand for humans.

## 1.3 Assembly languages (second generation)

Assembly languages were developed to make the instructions more readable for humans. Therefore, the sequences of bits were replaced by symbolic codes (usually based upon English like abbreviations). Unfortunately this results in code that is incomprehensible to computers until translated to machine language. The translator programs needed to accomplish this task are called assemblers.

Translating the example of figure 1 in assembly language results in:

```
LDA    060    (load content at memory address 0x60)
ADC    061    (add with carry to the content of memory address 0x61)
STA    062    (store result in memory address 0x62)
```

**Figure 3: example of assembly language code**

## 1.4 High-level languages

One of the disadvantages of assembly languages is that each assembly language instruction corresponds to exactly one machine language instruction. As such, programs written in assembly can easily become very long.

To speed up the process of writing programs, several high-level languages were developed. In all of these languages, single statements can accomplish substantial tasks. The syntax used resembles normal written language (mostly English), but with very strict regulations.

A program written in a high-level language is incomprehensible to a computer until translated into machine language. Depending on the high-level languages used, this translation is accomplished with a compiler or an interpreter. A compiler translates the complete source program at once. An interpreter translates one instruction at the time and executes that instruction directly.

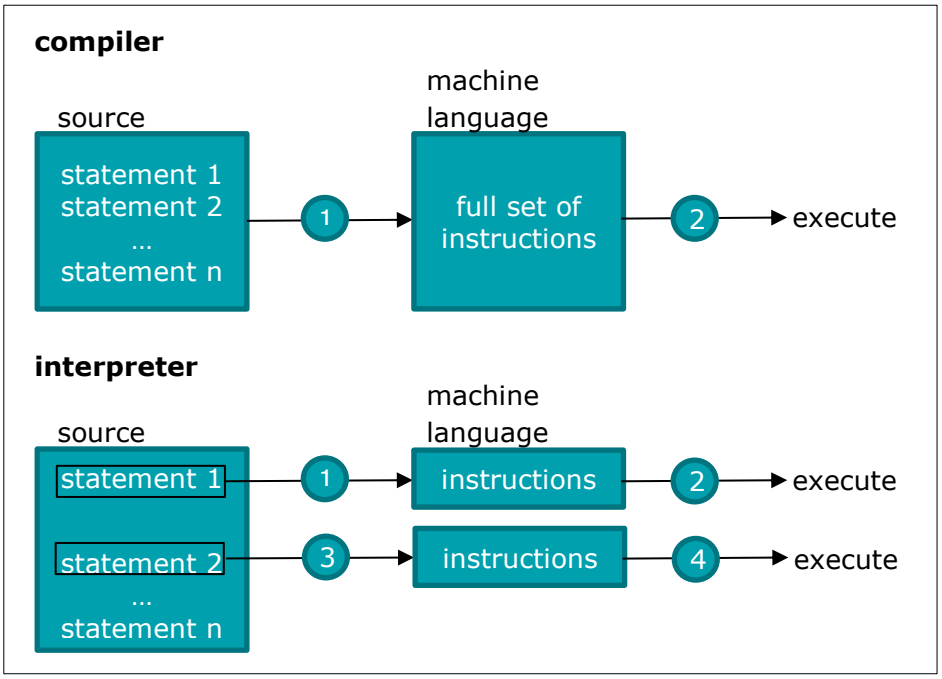



**Figure 4: difference between compiler and interpreter**

Some examples of high-level languages:

- FORTRAN (FORmula TRANslator, 1956):
  very efficient for mathematical computations
- COBOL (Common Business Oriented Language):
  used mainly for administration and accounting programs
- BASIC (Beginners All-purpose Symbolic Instruction Code, 1965): very straight forward, but not well structured
- PASCAL (Niklaus Wirth, 1970, named after Blaise Pascal):
  well-structured language, used mainly for educational purposes.

- C (Dennis M. Ritchie and Brian W. Kerninghan, 1972):
  well-structured and very efficient programming language available for most types of machines. It is widely used to develop systems that demand high performance, such as operating systems, embedded systems, real-time systems and communication systems as well as application software.
- LISP (LISt Processing Language): for "artificial intelligence"
- C++ (B. Stroustrup, 1986): C + object-oriented (OO = Object Oriented)

## 1.5 Processing a high-level language program

High-level language programs typically go through different phases to be executed.

### 1.5.1 Phase 1: creating a program

A standard text-editor is used to enter the wanted code into an ascii file. This file is commonly called the source code.

In this course, we will use the development environment "Visual Studio Express 2013 for Desktop" (see attachment 1). This environment contains a build-in editor that allows for easy program writing. All C program source codes should have names ending in .c (e.g.: `program1.c`)

### 1.5.2 Phase 2: translate the source code into machine language

In the second phase, the source code is compiled (translated) into machine language instructions resulting in an object code (e.g.: `program1.obj` or `program1.o`).

In a C system, this step is preceded by the execution of the preprocessor. The preprocessor searches for special directives written in the source code and performs the actions described. (see chapter 13 for more information on preprocessor directives).

After the execution of the preprocessor, the source code is translated into machine language instructions. When the compiler cannot recognize a certain statement, because of a syntax error for instance, the compiler issues an error message. These types of errors are called syntax errors or compile errors and must be corrected by editing the source code.

### 1.5.3 Phase 3: linking

All library functions referred inside the source code are added to the object file to create an executable program. (e.g.: `program1.exe`). This process is called linking and is performed by a linker. Errors occurring during this step are called link errors (such as calling non existing functions). Also in this

case, the source code needs to be corrected and all steps starting from phase 1 have to be executed again.

### 1.5.4 Phase 4: execution

Before execution, the program must be placed into memory. This is done by the loader. Additional components from shared libraries are loaded into memory as well.

Finally, the computer can execute the program. Errors occurring at this stage are called run time errors. These errors are caused by mistakes in the program logic like e.g. errors in the solution method used.

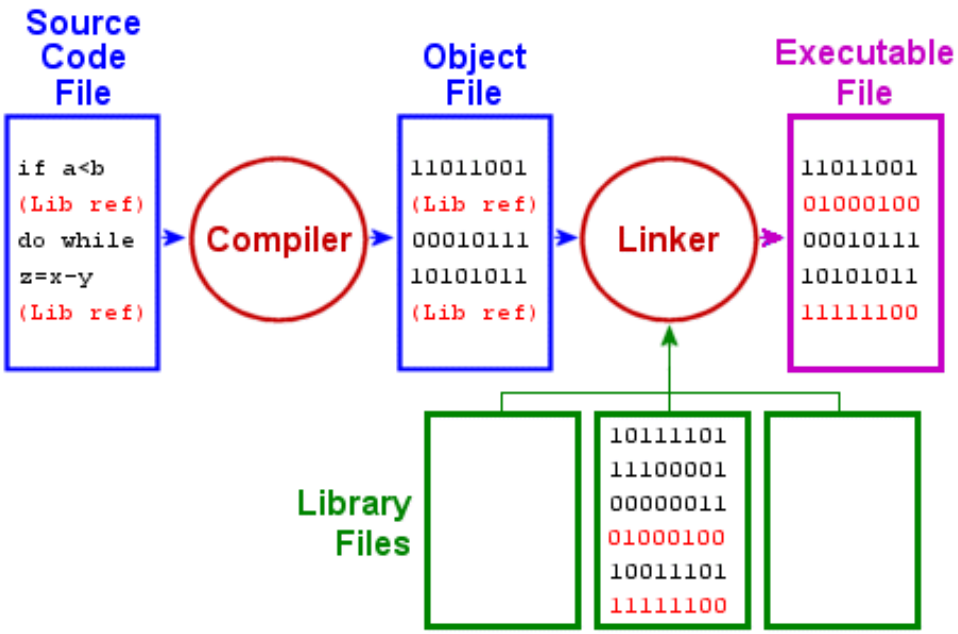



**Figure 5: processing flow for high-level language programs**

# 2 Program design

**Objectives**

In this chapter, the importance of thinking before coding is emphasized. The concept of an algorithm and how to use such an algorithm to write a computer program is explained. Special care was given to understanding the normal execution order and learning how to change that order in a well-structured way.

## 2.1 Algorithms

Writing computer programs nails down to writing algorithms in a computer language.

An algorithm is a step-by-step procedure that describes how to reach a predefined goal starting from an initial state and initial input in a finite time slot

Some well know algorithm examples are:

- a cooking recipe
- a construction manual
- installation instructions
- solve a set of equations

All of these algorithms consist of 3 phases:

1. description of the initial state and initial inputs
   e.g. ingredients needed for the cake
2. step-by-step description of the procedure to be followed
   e.g. how and in what order you need to mix the ingredients
3. description of the desired final state
   e.g. the cake itself

The language used to describe these algorithms is often incomprehensible to a computer. It needs to be translated into a programming language and afterwards compiled into computer language. An algorithm, written in programming language, is called a program.

## 2.2 Structured program development

There is more to programming than simply writing code! It involves all activities needed to write a computer program to solve a particular problem:

1. Description of the problem (what exactly needs to be done?)
2. Problem analysis

3. Divide into smaller sub problems
4. Write an algorithm for every sub problem
5. Translate these algorithms into code
6. Test all parts separately
7. Combine all sub codes and test the full code

Only the process of translating the algorithms into code is language dependent! Very often this is even the easiest part. Therefore, learning how to program in C also includes learning how to develop correct and efficient algorithms.

Think first, code later!

**Figure 6: think first, code later**

## 2.3 Documentation

In the process of solving a problem, documentation writing is often considered as a burden. However, it is important to make sure that after some time you or any other programmer can still easily understand and modify the code.

To accomplish this, following rules should be taken into account:

- use meaningful names
- use clear arrangement of the code (use the correct indentation)
- write appropriate, enlightening comments
- clearly indicate the different parts of the code and describe their function
- write basic documentation containing:
    - version, date, programmer
    - description of the problem
    - examples of execution
    - user manual

## 2.4 Program

A computer program is a sequence of instructions that are to be interpreted and executed by a processor. These instructions are executed one after the other in the order in which they’re written. This is called sequential execution.

At some point, a different execution order might be needed. Therefore, some control statements will be needed.

In modern programming, the concept of well-structured coding has become very important. To accomplish this, we will make use of 3 types of statements:

- sequential statements (normal execution order)
- selection statements (conditional execution of a group of statements)
- repetition statements (repeat a group of statements a number of times)

# 3 Programming in C: an introduction

**Objectives**

In this chapter some basic concepts of C programming are highlighted based on a few simple programming examples. At the end of this chapter you should have a basic understanding of:

- the structure of a C program
- variables and variable declaration statements
- assignment statements
- the usage of the standard functions `printf` and `scanf`
- the existence of control statements like `if` and `while`

## 3.1 A first program

We begin by writing a simple program that prints the text "Hello, world" on the screen. For more information on how to setup a first project see attachment 1. The program code and screen output are shown in Code 1.

```
1   /*
2      My first program
3      this program will print the line "Hello, world" to the screen
4
5      written by SBE 15/12/2014
6   */
7
8   #include <stdio.h>
9
10  int main(void)
11  {
12     printf("Hello, world\n");
13     return 0;
14  }
```


**Code 1: a first C program**

### 3.1.1 Comments

All text placed in between `/*` and `*/` is regarded as comments. The C compiler will ignore this portion of the code.

Since C is very well suited to write big programs, it is very important to make sure every part of the program is started with meaningful comments. These should at least indicate how the code works and for sure describe the outcome of that portion of the program.

In this example, a title, the outcome of the program and some information on author and creation date were written into the file as comments.

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

Every line starting with `#` is a preprocessor directive. Before compiling, the preprocessor will look for these directives and process them first. "`#include <stdio.h>`" tells the preprocessor to include the header file "`stdio.h`" into the program. This file contains some function declarations (not definitions) for standard input/output functions like `printf`, `scanf`, … The function definitions are located in the C standard library. The linker will take care of including all used functions into the executable. Next to function declarations, this header file also contains some definitions of constants (e.g. EOF, NULL) and macro's.

### 3.1.3 `int main(void){ }`

Every C program contains one or more functions. A function can be recognized by the parentheses () following the function name. One of those functions must be the function `main()`. Every C program will automatically start by executing this main function.

When calling a function, extra information can be passed on to that function. This extra information is called a function parameter and is put in between the parentheses (). To indicate that the main function does not have any parameters, the keyword `void` is used.

Functions can also return a result. The keyword `int` to the left of `main` indicates that the main function returns an integer value.

The full content of the main function is written in between an opening brace ( `{` ) and a closing brace ( `}` ). Every step of the function is described in a statement. The full collection of statements in between braces is called a block.

### 3.1.4 `printf("Hello, world\n");`

The main function in this example contains 2 statements: "`printf("…");`" and "`return 0;`".

A statement consists of an expression, followed by a semicolon (;). In C, every statement, even the last statement of a function, ends in a semicolon. There is no semicolon after "`int main()`" because here we start the definition of the function main, hence "`int main()`" is not a statement. However, if a function call is made, a semicolon needs to be added at the end.

The statement used here is

```
printf("Hello, world\n");
```

this is a call to the standard function `printf`. It will write all text in between the double quotes (“”) literally to the screen, except if an escape character (\) is used. When such an escape character is encountered in a string, the compiler takes the next character into account and combines it with the backslash to determine what to do. `\n` for instance means a newline. Therefore adding `\n` to the string in the `printf` function causes the cursor to position itself at the beginning of the next line.

The statement “`return 0;`” is added at the end of the `main()` function to set the return value of that function to the integer value 0.

## 3.2 Example 2

Convert 10 miles into km and print the result to the screen.

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5      float km, miles;
6      miles = 10.0;
7      km = miles * 1.609;
8      printf("%f mile equals %f kilometer\n", miles, km);
9      return 0;
10  }
```


**Code 2: code to convert 10 miles into km**

Explanation:

| | |
|---|---|
| `float km, miles;` | variable definitions |
| `miles = 10;`<br>`km = miles * 1.606;` | assignment statements |
| `printf("%f … %f …\n", miles, km);` | standard function `printf` |

### 3.2.1 Variable definitions

To store all sorts of values for use by the program, we need to easily address different memory locations. In C, this can be obtained by the use of variables. A variable is a symbolic name used to address a certain memory location. The variables used in the programming example above are `km` and `miles`.

All variables need to be defined with a name and data type before they can be used in a program. In this example, we use 2 variables of the type `float`, which means that they will hold real numbers.

### 3.2.2 Assignment statements

An assignment statement consists of two parts: an expression at the right-hand side and a variable that will hold the result of that expression at the left-hand side.
The statement "`miles = 10;`" indicates that the value `10` must be stored in the variable with name `miles`.

The statement "`km = miles * 1.606;`" indicates that the value of the variable `miles` will be multiplied with `1.606`. Afterwards, the result will be stored in the variable `km`.

### 3.2.3 `printf("%f mile equals %f kilometer\n", miles, km);`

This is again a function call to the standard function `printf`. However, this time, `printf` has 3 arguments: `"%f mile equals %f kilometer\n"` , `miles` and `km`.

The first argument is called the format string. It contains some literal characters to be displayed and some format conversion specifiers (`%f`). Every format conversion specifier starts with `%`. The letter `f` stands for floating point number, indicating that a real number will be printed instead of the "`%f`" sign in the format string. The second argument "`miles`" specifies the value to be printed instead of the first occurrence of "`%f`". The second occurrence of "`%f`" will be replaced by the content of the variable "`km`".

The function `printf` takes care of converting the internal representation of the variable into the form requested by the conversion specifier (decimal form in this case) before printing it to the screen.

## 3.3 Example 3

Extend the previous code with the possibility to obtain a distance in miles from the user, convert it to km and print the result to the screen.

```
1   /*
2      Program that reads a distance in miles and converts it to km
3   */
4
5   #include <stdio.h>
6
7   int main(void)
8   {
9      float km, miles;
10
11     printf("Enter the number of miles: ");
12     scanf("%f%*c", &miles);
13     km = miles * 1.609;
14     printf("%f mile equals %f kilometer\n", miles, km);
15     return 0;
16  }
```


**Code 3: read a number of miles and convert it to km**

**Explanation:**

The statement

```
scanf("%f%*c", &miles);
```

uses the standard function `scanf` to obtain a value from the user. This function reads from the standard input which is usually the keyboard (through a command line interface).

In this example, `scanf` has 2 arguments: a format string (the first argument) and a memory location. The `%f` conversion specifier indicates that the data entered by the user will be a real number. `scanf` will wait until the user enters a decimal number, convert it to the correct internal representation and store it to the memory location indicated by the second argument.

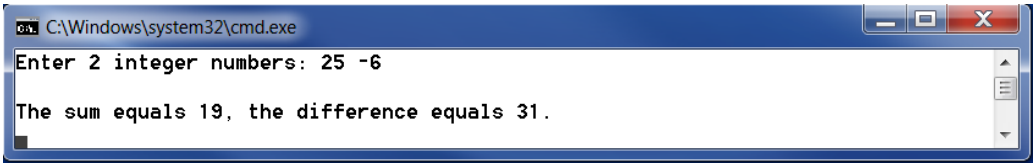
The second argument begins with an ampersand (`&`) followed by the variable name. The `&` in this expression means "address of". In the above example "`&miles`" indicates the address or the location in memory where the variable `miles` is stored.

The last part of the format string (`%*c`) indicates that yet another character (`%c`) needs to be read and disregarded (`*`). Adding "`%*c`" at the end of the format string allows to remove the enter from the input-buffer.

## 3.4 Example 4

Write a program that obtains 2 integer numbers from the user and prints the sum and difference of those two numbers to the screen.

```
1   /*
2      example4.c
3      this program computes sum and difference of two integer numbers
4   */
5
6   #include<stdio.h>
7
8   int main(void)
9   {
10     int number1, number2, sum, difference;
11     printf("Enter 2 integer numbers: ");
12     scanf("%d%d%*c", &number1, &number2);
13     sum = number1 + number2;
14     difference = number1 - number2;
15     printf("\nThe sum equals %d, the difference equals %d.\n",sum,
16            difference);
17     return 0;
18  }
```




**Code 4: compute the sum and difference of 2 integer numbers**

**Explanation:**

The statement

```
scanf("%d%d%*c", &number1, &number2);
```

uses the standard function `scanf` to obtain 2 integer values from the user. The `%d%d%*c` conversion specifiers indicate that the data entered by the user will be a sequence of 2 integer numbers, followed by an enter that does not need to be stored in memory. The first integer number read by the program, will be stored at the memory location indicated by `&number1` (address of the variable number1), the second number at the location referred to by `&number2`.

## 3.5 Example 5

This example shows a program that obtains two integer numbers from the user, computes their greatest common divisor (gcd) and prints it to the screen. For the computation of the gcd we will use the algorithm of Euclid:

1. Store the 2 integer numbers in the variables x and y such that x < y
2. Compute the remainder after y is divided by x
3. Store the value of x in the var y and the remainder in the var x
4. Repeat steps 2 and 3 until the remainder equals 0
5. The variable y contains the gcd

Example: compute the greatest common divisor of 34017 and 16966

- the remainder of 34017/16966 equals 85
- the remainder of 16966/85 equals 34
- the remainder of 85/34 equals 17
- the remainder of 34/17 equals 0
- the greatest common divisor of 34017 and 16966 equals 17

```
1   /* computation of the gcd of 2 integer numbers */
2
3   #include <stdio.h>
4
5   int main(void)
6   {
7      long int number1, number2;
8      long int x, y, remainder;
9
10     printf("Computation of the greatest common divisor.\n");
11     printf("Enter 2 positive integer numbers: ");
12     scanf("%ld%ld%*c", &number1, &number2);
13
14     /* put the smallest integer in x, the biggest one in y*/
15     if (number1 < number2)
16     {
17            x = number1;
18            y = number2;
19     }
20     else
21     {
22            x = number2;
23            y = number1;
24     }
25
26     /*assign a starting value to the variable remainder*/
27     remainder = x;
28
29     while (remainder != 0)
30     {
31            remainder = y % x;
32            y = x;
33            x = remainder;
34     }
35
36     printf("gcd(%ld, %ld) = %ld \n", number1, number2, y);
37     return 0;
38  }
```


**Code 5: computation of the gcd of 2 positive integer numbers**

**Explanation:**

This example contains 2 new types of statements: an `if` and a `while` statement. Both are examples of control statements.

Control statements can be used to alter the sequential execution of a program. Here, the `if` statement is used to determine the smallest of the 2

numbers and store it in the variable x. The while statement is used to translate the 4th step in Euclid's algorithm. It repeats the computation steps until the remainder equals 0.

Control statements will be treated in depth in chapter 5.

# 4 Basic concepts of C programming

**Objectives**

The basic concepts of C programming, highlighted in the previous chapter are explained in depth. At the end of this chapter you should have a good understanding of:

- indentation rules
- naming rules
- variables and variable declaration and initialization statements
- expressions
- assignment statements
- typecasting
- simple input/output functions

## 4.1 Indentation

C is a free format language. The compiler does not take white spaces into account, meaning that newlines and whitespaces can be inserted wherever wanted. However, to keep the code easily readable to everyone, we will follow some basic formatting rules.

Braces group statements together. Every statement inside the braces should be indented 1 tab (= 4 spaces). Both the opening and closing braces must be placed straight under the first letter of the function they belong to (see Figure 7).

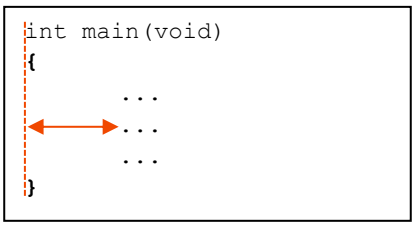



**Figure 7: indentation**

## 4.2 Identifiers

Every program needs a number of names or identifiers like names of variables, functions, …

All identifiers need to comply with following rules:

- all identifiers start with a letter
- identifiers consist of letters, digits and/or underscores ( _ )
- in ANSI-C, only the first 31 symbols of a name are significant
- C is case sensitive!
- Keywords (see table below) cannot be used as identifiers

| Keywords | | | | |
|---|---|---|---|---|
| auto | do | goto | signed | unsigned |
| break | double | if | sizeof | void |
| case | else | int | static | volatile |
| char | enum | long | struct | while |
| const | extern | register | switch | |
| continue | float | return | typedef | |
| default | for | short | union | |

## 4.3 Variables

### 4.3.1 Concept

A variable is a symbolic name used to address a certain memory location. The compiler will replace all variable names with their corresponding memory addresses.

A variable has a memory location (lvalue) indicated by the variable name and a value (rvalue). The variable in the example in Figure 8 has the name `number` and a value of 10.

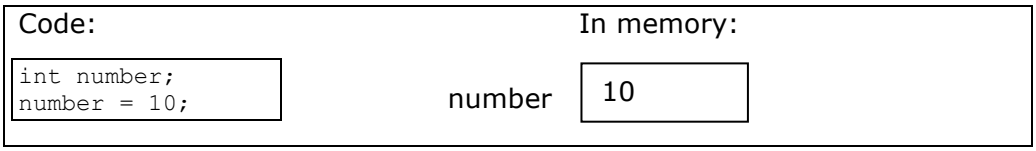



**Figure 8: lvalue and rvalue of a variable**

As a variable name, the programmer can choose any valid identifier (see 4.2).

**Learning note**

Choosing meaningful variable names helps to make the program more understandable. A meaningful name can save a few lines of comments.

### 4.3.2 Variable definition and data types

All variables need to be defined with a name and data type before they can be used in a program. The data type will determine the number of bytes needed to store that variable in memory and the type of operations allowed on it. In the example of Figure 8, the variable with name `number` is defined to be of the type `int`, which means that it will hold integer numbers. Depending on the system you work on, the amount of bytes needed to store integer values will be 2 or 4.

**Learning note**

Place the variable definitions immediately after the left brace that starts the (main) function. As such, all variables are automatically defined before being used.

Table 1 shows an overview of the most commonly used types in C along with the number of bytes needed and the allowed minimum and maximum values.

| type | size (bytes) | value | minimum | maximum | ex |
|---|---|---|---|---|---|
| char | 1 | symbols | 0 | 255 | 'a'<br>'A'<br>'\n'<br>'0' |
| short int | 2 | integer numbers | -32 768 | 32 767 | 32,<br>0x32,<br>0123 |
| int | 2 | integer numbers | -32 768 | 32 767 | 32,<br>0x32,<br>0123 |
| | or 4 | integer numbers | -2 147 483 648 | 2 147 483 647 | 32,<br>0x32,<br>0123 |
| long int | 4 | integer numbers | -2 147 483 648 | 2 147 483 647 | 32,<br>0x32,<br>0123 |
| float | 4 | real numbers | 3.4e-38 | 3.4e38 | 12.3<br>.5<br>5e4 |
| double | 8 | real numbers | 1.7e-308 | 1.7e308 | 12.3<br>.5<br>5e4 |

**Table 1: overview of the most commonly used data types in C**

**Remark**

Internally, the integer constant 0 is represented by 0000 0000 whereas the character '0' is represented by 0011 0000 (see ASCII table in attachment 2)

### 4.3.3 Variable initialization

Once a variable is defined, it corresponds to a certain memory location that is big enough to hold the specified type of data. However, the variable does not have a value (rvalue) yet. To assign a value to the variable, an assignment statement is needed.

In some cases, the value of the variable needs to be set before it can be used in a computation. The variable needs to be initialized. In the example of Figure 8, the variable `number` is initialized with the integer number 10 by the statement: "`number=10;`". After execution of this statement, the value 10 will be stored at the memory location the variable `number` is assigned to.

In C, a variable can be defined and initialized in one statement:

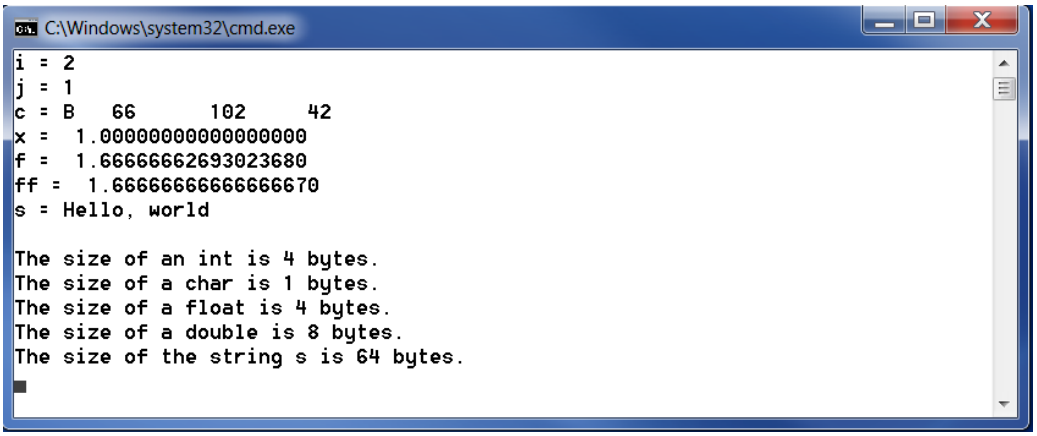
```
int number = 10;
```

**Remark**

Whenever a value is placed in a memory location, this new value replaces the previous value in that location!
Whenever a value is read from a memory location, the content of that location remains intact.

### 4.3.4 Example: variables.c

```
1   #include <stdio.h>
2   #include <string.h>
3
4   int main(void)
5   {
6      int i, j;
7      char c;
8      float x, f;
9      double ff;
10     char s[64];          // s can contain a series of chars (=string)
11     i = 2;
12     j = 5 / 3;
13     c = 'A';             // 0100 0001
14     c = c + 1;
15     x = 5 / 3;
16     f = 5.0 / 3;
17     ff = 5.0 / 3;
18     strcpy(s, "Hello, world");
19     printf("i = %d\n", i);
20     printf("j = %d\n", j);
21     printf("c = %c      %d     %o     %x\n", c, c, c, c);
22     printf("x = %20.17f\n", x);
23     printf("f = %20.17f\n", f);
24     printf("ff = %20.17f\n", ff);
25     printf("s = %s\n", s);
26     printf("\n");
27     printf("The size of an int is %d bytes.\n", sizeof(int));
28     printf("The size of a char is %d bytes.\n", sizeof(char));
29     printf("The size of a float is %d bytes.\n", sizeof(float));
30     printf("The size of a double is %d bytes.\n", sizeof(double));
31     printf("The size of the string s is %d bytes.\n", sizeof(s));
32     return 0;
33  }
```




**Code 6: example variables.c (code and screen output)**

Some explanation:

1. `j = 5 / 3;`
   the variable j is defined as type `int`. Therefore only integer values can be stored in this variable. As the division of 5 by 3 equals 1.667, only the integer part 1 is stored into variable j.

2. `strcpy(s, "Hello, world");`
   This is a function call to the standard function `strcpy` that copies the constant string "`Hello, world`" into the string variable `s`. The preprocessor directive "`#include <string.h>`" was added at the top to include the definition of the function `strcpy` into the program.

3. `printf("c = %c %d %o %x\n", c, c, c, c);`
   This statement prints the variable `c` in 4 different formats. First the value of `c` is printed as a character (`%c`), secondly as an integer number in decimal format (`%d`). The conversion specifier `%o` is used to print `c` as an integer number in octal format and finally `%x` obtains the hexadecimal format of the same integer number.

4. `sizeof`
   C provides the operator `sizeof` to determine the size in bytes of a data type.

## 4.4 Expressions

### 4.4.1 Arithmetic expressions

In most C programs, one or more calculations will be needed. To this end, the C language is equipped with a number of arithmetic operators as summarized in Table 2.

| operation | operator | example |
|---|---|---|
| addition | + | a + b |
| subtraction | - | a - b |
| multiplication | * | a * b |
| division | / | a / b |
| remainder | % | a % b |
| increment | ++ | i++ or ++i |
| decrement | -- | i-- or --i |

**Table 2: overview of arithmetic operators in C**

Some remarks:

- the division of two integer numbers yields an integer number:
  e.g. 1 / 2 = 0 but 1.0 / 2 = 0.5
- the remainder operand % yields the remainder after integer division:
  e.g. 5 % 3 = 2
  As such it can only be used with integer numbers.
- the increment (decrement) operand increments (decrements) the content of the variable with 1. Writing ++ (--) before or after the variable yields different results:
  e.g. what is the value of i and j in following code examples?

| | |
|---|---|
| `i=1;`<br>`j=1;`<br>`j=i++` | `i=1;`<br>`j=1;`<br>`j=++i;` |
| yields | |
| `i=2;`<br>`j=1;` | `i=2;`<br>`j=2;` |
| explanation | |
| Use the current value of i in the expression (j=i), afterwards increment i by 1. | First increment i by 1, then use the new value of i in the expression (j=i) |

**Figure 9: difference between i++ and ++i**

**Common mistake**

Attempting to divide by 0 results in a run time error that causes the program to stop.

### 4.4.2 Conditional expressions

Conditional expressions evaluate to one of following results:

- true (every value different from 0 is seen as true)
- false (0)

Conditional expressions are formed by using the equality operators, the relational operators and the logical operators as summarized in Table 3.

| operator | example | meaning of example |
|---|---|---|
| equality operators | | |
| == | x == y | x is equal to y |
| != | x != y | x is not equal to y |
| relational operators | | |
| < | x < y | x is less than y |
| <= | x <= y | x is less than or equal to y |
| > | x > y | x is greater than y |
| >= | x >= y | x is greater than or equal to y |
| logical operators | | |
| ! | !x | not x (logical not) |
| && | x && y | x and y (logical and) |
| \|\| | x \|\| y | x or y (logical or) |

**Table 3: equality, relational and logical operators**

The above operators can be combined into complex conditional expressions as shown by the examples below.

Algebra: $x \in ]2.0, 5.2]$ *or* $2.0 < x \leq 5.2$
C: $2.0 < x$ && $x <= 5.2$

Explanation: Condition to test for a leap year
C: $year\%4 == 0$ && $year\%100 != 0$ || $year\%400 == 0$

**Common mistake**

The symbols in the operators '==', '!=', '<=' and '>=' cannot be separated by one or more spaces!

**Common mistake**

Often the equality operator '==' is confused with the assignment operator '=' .
The expression 'x=5' is always true since the assignment operator takes care of assigning the value 5 to the variable x which is not equal to 0.
The expression 'x==5' is true only if the variable x contains the value 5.

Some expressions can cause run time errors under certain specific conditions only. E.g. the test '$t/n > 1$' will cause the program to fail whenever the variable n becomes equal to 0.

In C, these type of run time errors can be avoided using complex conditional expressions. If more than one operand is used, the evaluations will be done from left to right. Expressions further down the test will only be evaluated if needed.

To make the test '$t/n > 1$' error proof, it can be rewritten to:

$n! = 0$ && $t/n > 1$ or $n == 0$ || $t/n > 1$

### 4.4.3 Precedence

If an expression contains more than one operator, C will apply these operators in a sequence that is determined by the rules of precedence as summarized in Figure 10. The operators that were not treated yet will be explained in later chapters.


| |
|---|
| ( ) [ ] -> . |
| ! ~ ++ -- - (type) *(content) &(address) sizeof |
| * / % |
| + - |
| << >> (shift) |
| < <= > >= |
| == != |
| & (bit-AND) |
| ^ (bit exclusive OR) |
| \| (bit OR) |
| && (AND) |
| \|\| (OR) |
| ?: (conditional operator) |
| = += -= *= /= %= |
| , (comma-operator) |

**Figure 10: rules of precedence for operators in C**

In case two or more operators with the same level of precedence occur in one expression, the associativity of the operators is used. Most operators associate from left to right:

$1 + 2 - 3 + 4 - 5 \quad equals \quad \Big(\big((1 + 2) - 3\big) + 4\Big) - 5$

However, C contains some operators that associate from right to left:

- unary operators
  (operators that have only 1 operand like: !, unary +, unary -, ++, --)
- assignment operators
  e.g. $a = b = 0 \quad is\ interpreted\ as \quad a = (b = 0)$
- conditional operator (?:)

**Learning note**

To avoid confusion about the computation order in a complex expression, use parenthesis () to group parts of the expression.

## 4.5 Assignment statements

### 4.5.1 The assignment operator '='

To change the value stored in a variable, an assignment statement is used. In its most general form, the assignment statement can be written as:

```
<variable> = <expression>
```

The expression on the right-hand side of this statement will first be evaluated into a result. Afterwards, this result will be assigned to the variable written on the left-hand side of the assignment operator.

Examples:

```
sum = sum + number;
y=2 * cos(x) + 4;
```

**Common mistake**

A calculation in an assignment statement must be at the **right** side of the '=' operator!

### 4.5.2 Arithmetic assignment operators

C provides a number of assignment operators for abbreviating assignment expressions. For example the statement

```
x = x + a;
```

can be abbreviated with the addition assignment operator '+=' as

```
x += a
```

Table 4 provides an overview of the arithmetic assignment operators.

| operator | example: abbreviated | example: standard | explanation |
|---|---|---|---|
| += | x += a | x = x + a | increase x with a |
| -= | x -= a | x = x - a | decrease x with a |
| *= | x *= a | x = x * a | multiply x with a |
| /= | x /= a | x = x / a | divide x by a |
| %= | x %= a | x = x % a | calculate the remainder of the division of x by a |

**Table 4: arithmetic assignment operators**

**Common mistake**

The arithmetic and the assignment operator cannot be separated by one or more spaces!

## 4.6 Typecasting

In C, an expression has a value and a type. If for example the variables x and y are defined to be of the integer type, the result of the expression `'x/y'` will also be an integer. More precisely, the integer division of x by y will be performed in which any fractional part is truncated. So, even if this result is stored in a variable of the type float, the fractional part of the division will be lost.

To avoid this truncation, it is sufficient to explicitly convert the variable x into a float. This can be achieved by typecasting the variable x as follows:

```
result = (float)x / y;
```

In the above example, `'(float)x'` results in a temporary floating point copy of the variable x. As a result, the division will no longer be an integer division and yields a floating point result.

Actually, the division will be performed only after one extra action by the compiler. Due to the explicit type conversion of the variable x, x and y are now of different types. Since in C arithmetic expressions can only be evaluated if all operands are of the same type, a temporary floating point copy of the variable y will be made automatically, resulting in a floating point division. The result of this division is then assigned to the variable `result`.

In general explicit type conversion can be achieved by preceding the operand to be converted by the wanted type in between parenthesis:

```
(type)operand
```

A code example showing the effect of typecasting can be seen in Code 7.

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5      int x, y;
6      float res1, res2;
7      x = 1;
8      y = 2;
9      res1 = x / y;
10     res2 = (float)x / y;
11     printf("res1 = %8.4f \nres2 = %8.4f\n", res1, res2);
12     return 0;
13  }
```


**Code 7: typecasting example**

## 4.7 Simple input and output

### 4.7.1 printf()

The standard function printf() is a very powerful function that allows formatted printing to the screen in a flexible manner. To this end, the function printf() can have an arbitrary number of arguments that can be divided into 2 groups: the format control string and the other arguments.

The printf() function has the form:

```
printf(format control string, argument1, argument2, …)
```

The format control string starts and ends with double quotes (“”). It contains some literal characters to be displayed, some format conversion specifiers and some escape sequences. Every format conversion specifier starts with %, every escape sequence starts with \ . For example:

```
printf(“The letter %c stands for %s\n”, ‘a’, “alpha”);
```

#### 4.7.1.1 Format conversion specifiers

The format conversion specifiers in the above example are “%c” and “%s”. The letter c stands for character, indicating that a character will be printed instead of the “%c” sign in the format control string. The letter s stands for string, indicating that a series of characters (=a string) will be printed instead of the “%s” sign in the format control string. The second argument ‘a’ specifies the value to be printed instead of the first format specifier “%c”. The second format specifier “%s” will be replaced by the constant string “alpha”. The function printf() takes care of converting the internal representation of the arguments into the form requested by the conversion specifier before printing it to the screen. Table 5 shows an overview of the most commonly used conversion specifiers.

| type | conversion specifier | argument printed as ... |
|---|---|---|
| short (int) | %hd | integer value, decimal notation |
| int | %d | integer value, decimal notation |
| int | %x | integer value, hexadecimal notation |
| long (int) | %ld | integer value, decimal notation |
| float | %f | real value, fixed point notation |
| float | %e | real value, scientific (exponential) notation |
| double | %lf | real value, fixed point notation |
| char | %c | individual character |
| string | %s | string of characters |

**Table 5: conversion specifiers**

#### 4.7.1.2 Escape sequences

Next to conversion speficiers, the format control string can also contain escape sequences. An escape sequence starts with the character '\' . When such an escape character is encountered in a string, the compiler takes the next character into account and combines it with the backslash to determine what to do. '\n' for instance means a newline. Therefore adding '\n' to the format control string in the `printf()` function causes the cursor to position itself at the beginning of the next line. Table 6 shows an overview of the most commonly used escape sequences

| escape sequence | description |
|---|---|
| \a | Alert (audible (bell) or visual alert) |
| \b | Backspace (cursor moves back one position) |
| \f | Formfeed (cursor moves to start of next logical page) |
| \n | Newline (cursor moves to beginning of next line) |
| \r | Carriage return (cursor moves to beginning of current line) |
| \t | Horizontal tab (cursor moves to next horizontal tab pos) |
| \v | Vertical tab (cursor moves to next vertical tab position) |
| \? | Question mark |
| \' | Single quotation mark |
| \" | Double quotation mark |
| \\ | Backslash |
| \xdd | ASCII character in hex notation |
| \0 | Null character |

**Table 6: escape sequences**

#### 4.7.1.3 Field widths and precision

The field width specifies the size of the field the data is printed in. This field width can be set by inserting an integer number between the percent sign (%) and the conversion specifier.

Examples:

| | |
|---|---|
| %8d | field width = 8 character positions,<br>if the printed integer is smaller than 8 positions, the integer will be right-aligned |
| %-8d | field width = 8 character positions,<br>if the printed integer is smaller than 8 positions, the integer will be left-aligned |
| %-40s | the printed string will be left-aligned in a total field width of 40 symbols. |

The function `printf` also enables you to set the print precision. To this end, a dot followed by an integer number must be inserted before the conversion specifier.

| | |
|---|---|
| `%16.8lf` | total field width = 16 character positions,<br>8 digits after the comma will be printed (leaving a maximum of 7 digits before the comma and one decimal point)<br>if the printed number is smaller than 16 positions, it will be right-aligned |
| `%.2f` | print with 2 digits after the decimal point |

### 4.7.2 `scanf()`

The function `scanf()` allows to read formatted input from the stdin (mostly the keyboard).

The `scanf()` function has the form:

```
scanf(format control string, argument1, argument2, …)
```

The format control string starts and ends with double quotes (“”). It contains a sequence of 1 or more format conversion specifiers to specify type and format of the data to be retrieved from stdin. This data is then stored in memory locations pointed to by the additional arguments.

As a result, all additional arguments in a `scanf` function must be addresses. This can be achieved by putting the operator `&` (= address of) before the name of the variable. Some variable names already represent an address (like names of strings and pointers). In those cases the `&` sign must be omitted.

Examples:

| | |
|---|---|
| `scanf(“%d%*c”,&number);` | read an integer number from the stdin and store it in the variable `number` |
| `scanf(“%f%f%*c”,&km,&miles);` | read two real numbers, store the first one in the variable `km` and the second one in the variable `miles` |
| `scanf(“%lf%*c”,&ff);` | read a real number and store it as a double in the variable `ff` |
| `scanf(“%s%*c”,s);` | read a string and store it in the variable `s`. Note that s is a string so it is already an address without adding the & operator. |

**Remark**

The '%*c' at the end of all format conversion strings in the above examples indicates that a character (in this case the <Enter>-symbol) must be read but not saved into a variable. In this way the <Enter>-symbol is cleared from the stdin buffer and can no longer be taken as input in a next `scanf` statement.

**Common mistake**

All additional arguments of the scanf function must be addresses. Using a variable name without `&` sign is a very common mistake! Be aware that some variable names already are addresses (like strings and pointers). These variables cannot be preceded by the `&` sign.

### 4.7.3 gets(), puts()

The function

```
gets(argument)
```

reads characters from stdin until a newline or `EOF` character is read and stores these characters as a string in the memory location the argument points to. This string does not contain the ending newline character. To make it a valid string, a null byte (`\0`) is automatically appended.

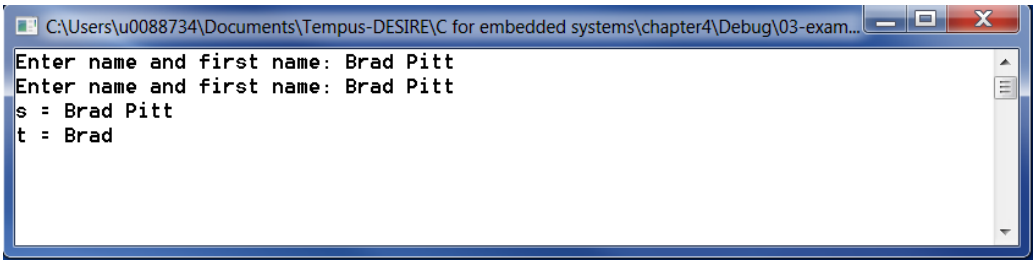
The function

```
puts(argument)
```

writes the string that is stored at the memory location the argument points to, to the screen followed by a newline character.

The following example shows the difference between the function `gets` and the function `scanf` when a string is to be read.

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5      char s[32];
6      char t[32];
7      printf("Enter name and first name: ");
8      gets(s);
9      printf("Enter name and first name: ");
10     scanf("%s", t);
11     printf("s = %s\nt = %s\n", s, t);
12     return 0;
13  }
```




**Code 8: reading strings with gets and scanf**

As can be seen in the example of code 8, the function `gets` reads a full line (all characters until a newline is read), whereas the function `scanf("%s", …)` only reads the first string (until a white space character or newline character is read) !

### 4.7.4 `getchar()`, `putchar()`

The function `getchar()` reads the next character from stdin and returns it as an integer.

The function `putchar()` writes 1 character to stdout.

## 4.8 Exercises

**4.1.** Write a program that asks the user to enter an integer number, stores this number in a variable of the type short (short int) and prints it back to the screen. What values can be entered? What is the biggest number you can enter and print correctly?

**4.2.** Repeat the previous exercise but this time use a variable of the type int or long.

**4.3.** Write a program that
- asks the user to enter 2 integer numbers in one line
- obtains those 2 numbers using only one call of the function scanf
- prints both numbers on the next line

the screen dialogue should look like:

```
Enter 2 integer numbers: 17  7
The entered numbers are 17 and 7
```

**4.4.** Write a program that asks the user to enter 2 real numbers and prints their sum and product to the screen.

**4.5.** Write a program that asks the user to enter 2 integer numbers and prints the result and remainder of the integer division of both numbers and the real quotient

the screen dialogue should look like:

```
Enter 2 integer numbers: 17  7
integer quotient = 2
remainder = 3
quotient = 2.43
```

**4.6.** Write a program that asks the user to enter 2 real numbers, calculates the quotient and stores it into a variable of the type float. Print the quotient with 20 numbers after the decimal point.

**4.7.** Write a program that asks the user to enter hours, minutes and seconds separately and prints them in a sentence like:
`The entered time is: hh hours mm minutes and ss seconds`
where hh and mm can only be integer numbers. ss can have digits after the decimal point

**4.8.** Write a program that asks the user to enter name and first name separately and prints them on 1 line. Test your program with names that contain white spaces (ex: Julia Rose Smith)

**4.9.** Write a program that asks the user to enter the radius of a circle, calculates the surface area of that circle and prints the result to the screen. ($\pi$ can be approximated as 3.141592653589793. Define $\pi$ as a constant in your program).

**4.10.** Write a program that asks the user to enter a number of seconds and prints the corresponding number of days, hours (<24), minutes (<60) and seconds (<60).
ex: 90061 sec = 1 day 1 hour 1 minute and 1 second

**4.11.** Write a program that asks the user to enter 5 integer numbers one by one. While reading the numbers, the program calculates the sum. At the end, the mean of all 5 numbers is printed with a precision of 2 digits after the digital point. Try to limit the number of variables in your program to 2 (or max 3) but do not introduce loops yet.

**4.12.** Write a program that asks the user to enter an amount of money (e.g. 13578) in euro and prints the corresponding number of notes (500, 200, 100, 50, 20, 10, 5) and coins (2, 1). Always use the minimal number of notes and coins possible.

**4.13.** Rewrite the previous program such that cents (50, 20, 10, 5, 2, 1) are also included (e.g. 13578,78). Be aware that the % operator can only be used with integer operands!

**4.14.** Consider an electrical circuit consisting of two series resistors R1 and R2. If a voltage U is applied to this circuit, the current flowing in this circuit will be $I = \frac{U}{R1+R2}$ according to Ohm's law. The voltage (V2) across the resistor R2 will then be defined by $2 = I\,.R2$ .
Write a program that asks the user to enter the values for U, R1 and R2 and prints the values of I and V2 to the screen.

**4.15.** Write a program that reads 6 integer numbers and prints them in a table format with 3 rows and 2 columns.

the screen dialogue should look like:

```
Enter 6 integer numbers: 1 22 33 4 5 6

 1     22
33      4
 5      6
```

**4.16.** Rewrite the previous exercise such that lines are placed around the table and in between the numbers:

```
Enter 6 integer numbers: 1 22 33 4 5 6

-------------
|   1 |  22 |
|-----------|
|  33 |   4 |
|-----------|
|   5 |   6 |
-------------
```

**4.17.** Rewrite the previous exercise with real numbers. Make sure the decimal points are nicely aligned.

**4.18.** Write a program that asks the user to enter name, first name, street, number, zip code and town name. Afterwards, the program outputs this data in the format:

```
first name   name
street  number
zip code         town
```

**4.19.** Write a program that asks the user to enter the invoice number, the number of products ordered and the price per unit. Calculate the total amount to be paid and print it to the screen.

the screen should look like:

```
Enter invoice number: 12
Enter the number of products ordered: 50
Enter the price per unit: 599

INVOICE      NUMBER      PRICE/UNIT        TOTAL
12           50          599               29950
```

**4.20.** Write a program that asks the user to enter an integer number with 3 digits and prints the number backwards.

```
Enter an integer number with 3 digits: 123
The number printed backwards is: 321
```

Hint: $321 = 3 * 100 + 2 * 10 + 1 \; with \; 1 = \frac{123}{100}$ , $2 = \frac{23}{10}$ $and \; 3 = remainder$

**4.21.** Write a program that prints the date of Easter for a year entered by the user.
Easter is held on the first Sunday after the first full moon of the spring. Easter is delayed by 1 week if the full moon is on Sunday. According to Jean Meeus, Spencer Jones and Butcher, the Easter date in year J can be calculated as follows (all divisions are integer divisions):

$a = remainder \; of \; the \; division \; of \; J \; by \; 19$
$b = \frac{J}{100}$
$c = remainder \; of \; the \; division \; of \; J \; by \; 100$
$d = \frac{b}{4}$
$e = remainder \; of \; the \; division \; of \; b \; by \; 4$
$f = \frac{b + 8}{25}$
$g = \frac{(b - f + 1)}{3}$
$h = the \; remainder \; of \; the \; division \; of \; (19 * a + b - d - g + 15) \; by \; 30$
$i = \frac{c}{4}$
$k = the \; remainder \; of \; the \; division \; of \; c \; by \; 4$
$l = the \; remainder \; of \; the \; division \; of \; (32 + 2 * e + 2 * i - h - k) \; by \; 7$
$m = \frac{(a + 11 * h + 22 * l)}{451}$
$month = \frac{h + l - 7 * m + 114}{31}$
$day = 1 + the \; remainder \; of \; the \; division \; of \; (h + l - 7 * m + 114) \; by \; 31$

This formula is valid for the Gregorian calendar and as such only after 1582.

Some test values:

| Year | Easter date |
|---|---|
| 2005 | March 27 |
| 2006 | April 16 |
| 2007 | April 8 |
| 2008 | March 23 |
| 2009 | April 12 |
| 2010 | April 4 |
| 2011 | April 24 |
| 2012 | April 8 |
| 2013 | March 31 |
| 2014 | April 20 |

# 5 Controlling the program flow

**Objectives**

In this chapter, the different control structures in C are explained. At the end of this chapter you should be able to use:
- flowcharts
- selection statements (if, if … else and switch) to select actions
- iteration statements (while, do … while and for) to repeat actions

Normally, statements in a program are executed one by one in the order in which they are written.

Rather often, this sequential execution needs to be broken. C provides two groups of control statements that allow to create all possible program flows. The first group consists of selection statements. They are used whenever a certain statement or group of statements can only be executed in well-defined conditions. Selection statements will be explained in section 5.2.

The second group contains the repetition statements and allows to repeat one or a group of statements a number of times. Repetition statements will be treated in section 5.3.

To visualize the wanted program flow, we will make use of flowcharts. This will be handled in section 5.1.

## 5.1 Flowchart

As indicated in chapter 2, it is important to think before coding. Therefore, every program writing process should start with the construction of an algorithm. Since a picture is worth a 1000 words, it is useful to first draw a graphical representation of that algorithm before translating the algorithm into C code. This does not only allow to clearly show the expected program flow, but also allows for faster detection of possible reasoning errors.

Different types of graphical representations exist in literature. In this course, flowcharts will be used.

Since flowcharts can serve different purposes, only a small subset of possible flowchart symbols will be used. Table 7 shows an overview of these symbols.

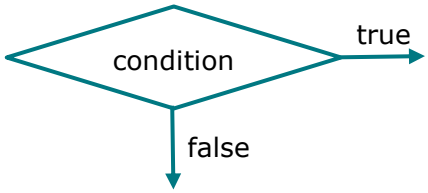
| symbol | description |
|---|---|
|  | Indicates the start or end of a program or function |
|  | represents an action or statement |
|  | represents a function call. The flowchart of that function can then be written separately. |
|  | represents a test for a certain condition. This condition can either be true or false (yes or no). The arrow representing true (false) can be either of the 2 arrows. |
|  | flow lines are used to connect the different symbols in the correct order. |

**Table 7: flowchart symbols**

In Figure 11 an example of a flowchart for a pure sequential program is shown.

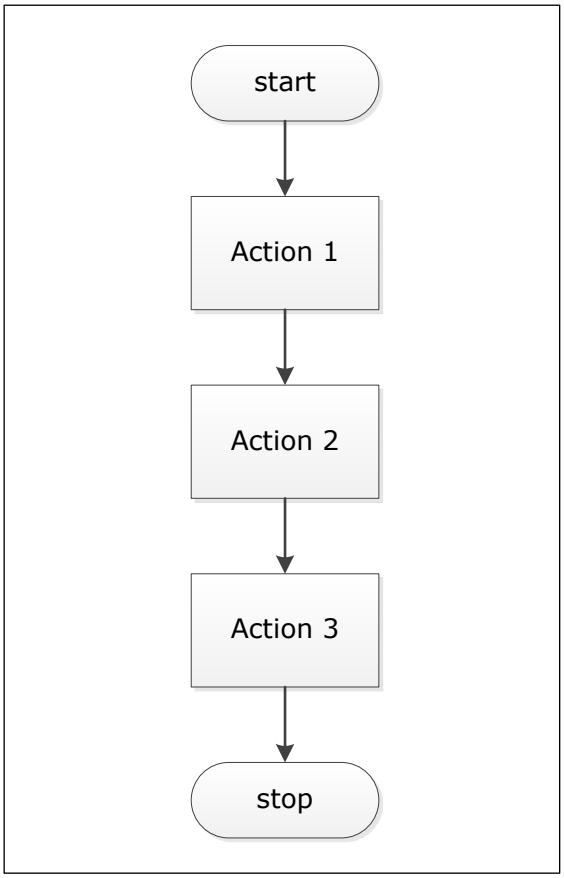



**Figure 11: flowchart sequential program**

## 5.2 Selection statements

C provides three types of selection statements. The first type is the `if` selection statement that either performs an action if a certain condition is met or skips the action otherwise. The second type is the `if … else` statement that allows to perform one out of two actions depending on whether the condition is met or not. The last type (`switch`) allows to select one of many possible actions.

### 5.2.1 The `if` selection statement

General form:

```
if (<expression>)
    <statement 1>;
<statement 2>;
```

if the expression between parenthesis yields a number different from 0 (=true), statement 1 is executed followed by statement 2. On the other hand, if the expression equals 0 (=false), statement 1 is skipped and statement 2 is executed directly.

If more statements are to be executed when a certain condition is met, the statements can be grouped into a code block:

```
if (<expression>)
{
    <statement 1>;
    <statement 2>;
}
<statement 3>;
```

The flowchart of the `if` statement is shown in Figure 12.

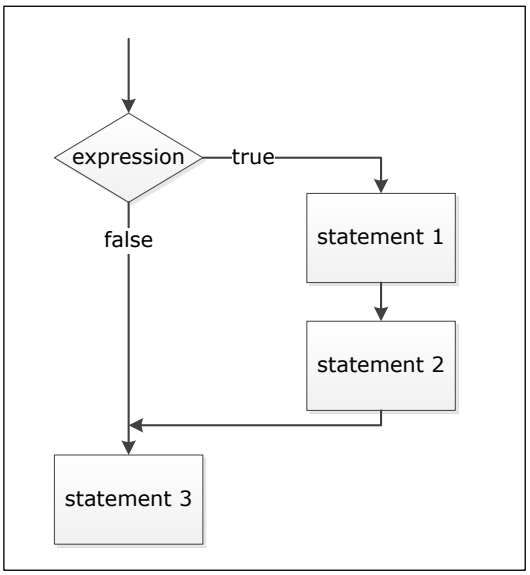



**Figure 12: flowchart `if` selection statement**

### 5.2.2 The `if … else` selection statement

#### 5.2.2.1 General form

```
if (<expression>)
    <statement 1>;
else
    <statement 2>;
<statement 3>;
```

if the expression between parenthesis yields a number different from 0 (=true), statement 1 is executed followed by statement 3. If the expression equals 0 (=false), statement 2 is executed followed by statement 3. Also in this case, the statements 1 and 2 can be block-statements:

```
if (<expression>)
{
    <statement>;
    <statement>;
    …
}
else
{
    <statement>;
    <statement>;
    …
}
```

The flowchart now looks like:

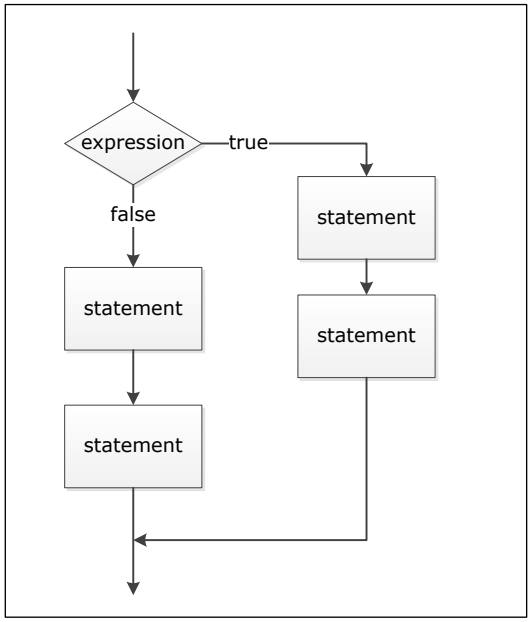



**Figure 13: flowchart `if ... else` statement**

**Learning note**

Note that the braces in the `if` statement and in the `if … else` statement are always placed right under the first letter of `if` (or `else`) and that all body statements are indented! This improves the readability of the code!

**Example**:
Write a C program that reads an integer number and prints either even or odd depending on the number read.

A first possible solution is represented by the flowchart in Figure 14. Of course also the flowchart in Figure 15 is valid for this example. Which one to choose is entirely up to the programmer.

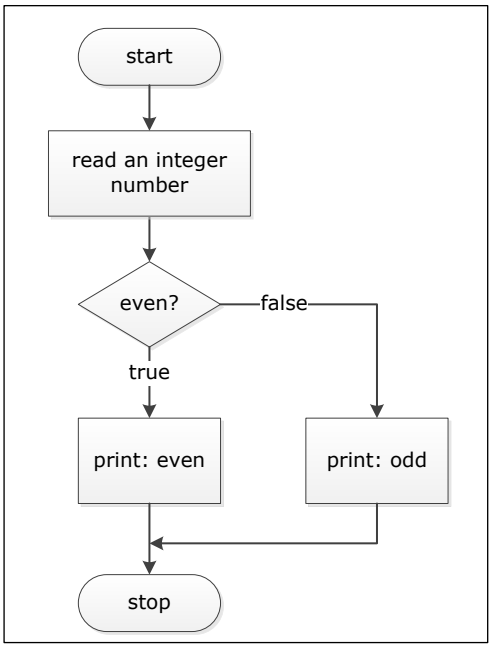



**Figure 14: even-odd solution 1**

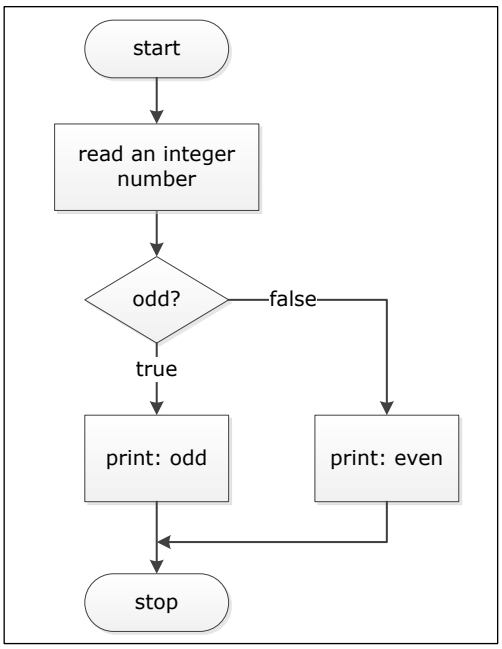



**Figure 15: even-odd solution 2**

Translating the flowchart of Figure 14 into C code yields:

```
1   /*
2      read an integer number and print odd or even depending on
3      the number read
4   */
5
6   #include <stdio.h>
7
8   int main(void)
9   {
10     int number;
11
12     printf("Enter an integer number: ");
13     scanf("%d%*c", &number);
14
15     if (number % 2 == 0)
16     {
17           printf("The number %d is even\n", number);
18     }
19     else
20     {
21           printf("The number %d is odd\n", number);
22     }
23
24     return 0;
25  }
```

**Code 9: example even - odd solution 1**

The test in the if statement should not necessarily be a logical expression. It can be any mathematical expression as shown in Code 10. The flowchart corresponding to this solution is represented in Figure 15.

```
1   /*
2      read an integer number and print odd or even depending on
3      the number read
4   */
5
6   #include <stdio.h>
7
8   int main(void)
9   {
10     int number;
11
12     printf("Enter an integer number: ");
13     scanf("%d%*c", &number);
14
15     if (number % 2)
16     {
17           printf("The number %d is odd\n", number);
18     }
19     else
20     {
21           printf("The number %d is even\n", number);
22     }
23
24     return 0;
25  }
```

**Code 10: example even - odd solution 2**

In the above code the expression "`number % 2`" equals 0 if the variable "`number`" contains an even value. So if an even number is entered, the statements belonging to `else` are executed.

**Common mistake**

Often the equality operator '==' is confused with the assignment operator '=' :

```
if (x = 5)
{
      The statements written here will always be executed! The
      expression 'x=5' is always true since the assignment
      operator takes care of assigning the value 5 to the
      variable x which is not equal to 0!
}
```

**Common mistake**

No semicolon (;) can be placed after the condition!

```
if (x == 5);
{
      These statements will always be executed since the ";"
      ends the if statement!
}
```

#### 5.2.2.2 The conditional operator (?:)

The conditional operator in C is closely related to the `if … else` statement. It takes 3 operands. The first one is a condition, the second one is the value for the entire conditional expression if the condition is true and the third one will be used as value for the entire expression if the condition is false.

General form:

```
<condition> ? <value if condition is true> : <value if condition is false>
```

Example:

```
if (a < b)
       z = a + 1;
else
       z = b - 1;
```

can be written with the conditional operator as:

```
z = a < b ? a + 1 : b - 1 ;
```

More examples:

```
printf("%4d%c", a[i] , ((i+1) % 10 ? ' ' : '\n' ) );

printf(number % 2 ? "odd" : "even");

printf("%d is ",year);
printf(year%4==0 && year%100!=0 || year%400==0? "a leap year\n" : "no
leap year\n");
```

#### 5.2.2.3 Nested `if .. else` statements.

The code-blocks inside the `if` statement can again contain an `if … else` statement. As such nested `if … else` statements are created:

```
if(expression1)
{
    statement;
    …
}
else
{
    if(expression2)
    {
         statement;
         …
    }
    else
    {
         statement;
         …
    }
}
```

**Learning note**

Watch the indentation of the different body statements! Make sure indentation is done correctly to avoid confusion!
An `else` in such a control structure belongs to the latest `if` that did not get an `else` yet. Correct indentation can help to quickly understand which `else` belongs to which `if` statement.

Following example shows the importance of correct indentation:

```
1   #include <stdio.h>
2   int main(void)
3   {
4      int number;
5      printf("Enter integer number: ");
6      scanf("%d%*c", &number);
7      if (number > 70)
8             if (number > 80)
9                    printf("passed with great honor!\n");
10     else
11            printf("ok.\n");
12  }
```

What do you expect if for instance the number 65 is entered?

The indentation chosen in the example above is misleading. It suggests that the result for a number smaller than or equal to 70, is the text "ok" printed to the screen. Instead, nothing will appear since the `else` that contains the print statement belongs to the last `if` without `else` being `if(number > 80)`
Correctly indenting the `else` results in:

```
1   #include <stdio.h>
2   int main(void)
3   {
4      int number;
5      printf("Enter integer number: ");
6      scanf("%d%*c", &number);
7      if (number>70)
8             if (number>80)
9                    printf("passed with great honor!\n");
10            else
11                   printf("ok.\n");
12  }
```

If the `else` must belong to the first `if`, brackets need to be added:

```
1   #include <stdio.h>
2   int main(void)
3   {
4      int number;
5      printf("Enter integer number: ");
6      scanf("%d%*c", &number);
7      if (number>70)
8      {
9             if (number>80)
10                   printf("passed with great honor!\n");
11     }
12     else
13            printf("ok.\n");
14  }
```

### 5.2.3 The `switch` statement

To select one of many actions, nested `if` statements can be used. If the choice for the right action can be made based upon the result of an integer expression, a `switch` statement is shorter and provides a clearer structure.

General form:

```
switch (<integer expression>)
{
    case <value1> :
       <0 or more statements>;
       break;
    case <value2> :
       <0 or more statements>;
       break;
    …
    default:
       <0 or more statements>;
}
<next statement>
```

The value of the integer expression is tested against the constant integral values of every `case`. At the first match, the corresponding statements are executed. If the last statement corresponding to that specific case is a `break;` the execution of the switch will stop and the `<next statement>` after the switch will be handled. If not, the statements belonging to the next `case` will be executed.

If no match is found, the default statements are executed. The default can be omitted. In that case, nothing will happen when no match occurs.

The switch statement can be represented by the flowchart of Figure 16.

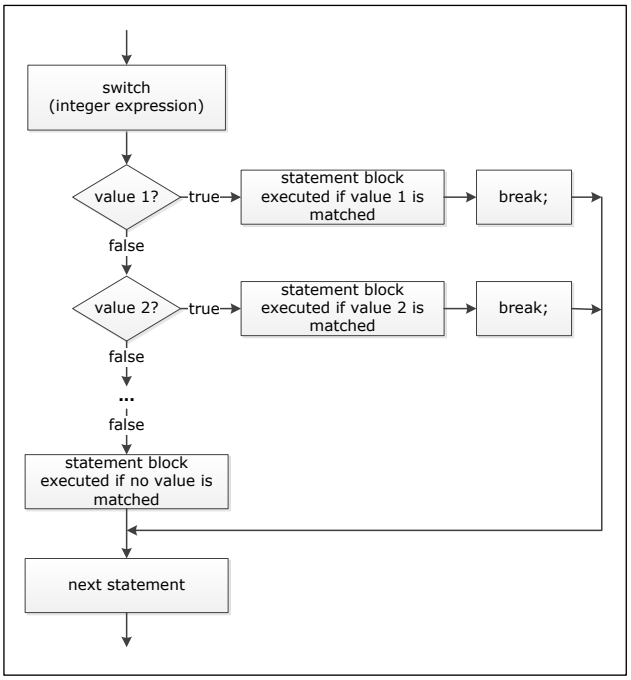



**Figure 16: flowchart switch statement**

**Common mistake**

Do not forget to put a `break` at the end of each case (unless the `break` is left out intentionally).

**Example**:
Write a C program that reads a letter and prints a country name depending on the letter read as follows: a/A = Argentina, b/B = Brazil, h/H = Honduras, m/M = Mexico, p/P = Peru. All other letters yield the sentence "Unknown country".

A possible solution is represented by the flowchart in Figure 17.

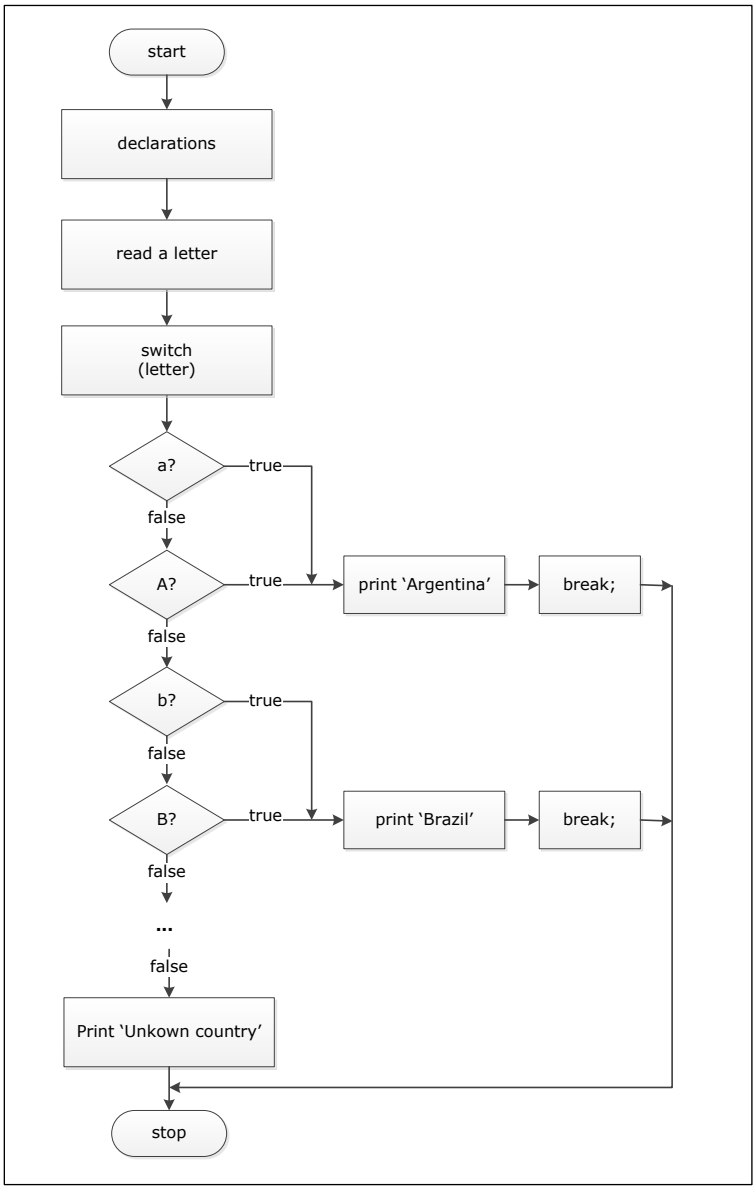



**Figure 17: flowchart `switch` example**

Translation into C code yields:

```
1   /*
2      read a letter and print the corresponding country
3   */
4   #include <stdio.h>
5
6   int main(void)
7   {
8      char symbol;
9
10     printf("Enter a letter: ");
11     symbol = getchar();
12
13     switch (symbol)
14     {
15     case 'a':                        // letter was lowercase a
16     case 'A':                        // or uppercase a
17            printf("Argentina\n");
18            break;                    // necessary to exit switch
19     case 'b':
20     case 'B':
21            printf("Brazil\n");
22            break;
23     case 'm':
24     case 'M':
25            printf("Mexico\n");
26            break;
27     case 'p':
28     case 'P':
29            printf("Peru\n");
30            break;
31     case 'h':
32     case 'H':
33            printf("Honduras\n");
34            break;
35     default:            // catch all other letters
36            printf("Unknown country\n");
37            break;       // optional since switch will exit anyway
38     }
39
40     return 0;
41  }
```

**Code 11: example of `switch`**

The `switch` in the above example can be written much shorter if we make use of the string function "`toupper()`" that converts a lowercase character into an uppercase character:

```
switch (toupper(symbol))
{
       case 'A':
           printf("Argentina\n");
           break;
       case 'B':
           printf("Brazil\n");
           break;
       ...
       default:
           printf("Unknown country\n");
           break;
}
```

**Code 12: alternative code for `switch` example**

### 5.2.4 Exercises

**5.2.1.** Write a program that asks the user to enter an integer number and prints whether that number is positive or not (= negative or 0!).

**5.2.2.** Write a program that asks the user to enter an integer number and prints whether that number is even or odd.

**5.2.3.** Write a program that asks the user to enter a real number and prints whether that number lies in [5,10[ or not ([5,10[ means $5 \leq x < 10$)

**5.2.4.** Write a program that asks the user to enter a real number and prints whether for that number, each one of the following conditions is met or not:

condition A: $3 \leq x < 8.5$
condition B: $x < 3$ OR $5.4 < x \leq 7.3$ OR $x > 13$
condition C: $x \neq 3$ AND $x < 9.75$

examples:
0 matches conditions: not A, B, C
3 matches conditions: A, not B, not C
8 matches conditions: A, not B, C
15 matches conditions: not A, B, not C

**5.2.5.** Write a program that asks the user to enter a real number and prints whether that number is positive, zero or negative.

**5.2.6.** Write a program that asks the user to enter 5 integer numbers one by one. While reading the numbers, the program calculates the minimum of those numbers. At the end, this minimum is printed. Try to limit the number of variables in your program to 2 (or max 3) but do not introduce loops yet.

**5.2.7.** Write a program that compares your speed with the speed limit. If you are speeding, the program will calculate your fine. If not, nothing happens. The fine consists of a fixed amount of € 100 and a variable amount of € 2.5 for every km over the speed limit.

The screen should look like:

```
Be aware! Speeding is heavily fined!
Enter your speed: 131
Enter the speed limit: 120
Your speed is 11 km/h over the speed limit.
Your fine amounts 127.5 euro.
```

**5.2.8.** Write a BMI (Body Mass Index) calculator. BMI is a measure of body fat based on height and weight that applies to adult men and women. It can be calculated with the following formula:

$$BMI = \frac{weigth\ in\ kg}{(heigth\ in\ m) * (heigth\ in\ m)}$$

The BMI is divided into different categories:
– underweight < 18.5
- normal weight 18.5 – 24.9
- overweight 25 – 29.9
- obesity ≥ 30

Ask the height and weight of the user, calculate his/her BMI and print the category the user belongs to.

**5.2.9.** Write a program that first reads 3 integer numbers that represent the current date and then reads again 3 integer numbers to be interpreted as a birth date. Based upon this information the program prints the age of that person in years and months.

```
Enter the current date: 6 1 2015
Enter your birth date: 25 1 1985
Your age is: 29 years and 11 months
```

**5.2.10.** Write a program that reads a start and end time, calculates the time difference and prints it in the format hh hours mm minutes ss seconds. You can read hours, minutes and seconds separately. If the start time appears to be later than the end time, you can assume the start time to be from the previous day.

examples:

| start | 2 12 12 | 3 12 18 | 5 23 45 | 21 0 0 |
|---|---|---|---|---|
| end | 3 15 17 | 3 15 17 | 7 10 30 | 4 30 15 |
| time difference | 1 3 5 | 0 2 59 | 1 46 45 | 7 30 15 |

**5.2.11.** Write a program that reads 3 numbers and prints them ranked form small to large.

**5.2.12.** Write a program that reads a digit (0, 1, … , 9) and prints this digit as a word. If the entered number is not a digit, a warning should be printed:

```
Enter a digit: 3
You have entered the digit three.
```

```
Enter a digit: 23
The number you entered is not a digit.
```

**5.2.13.** Write a program that asks the user to enter an instruction in the format:

number1 operand number2

For the operand, the user can choose +, -, * or /. The program calculates the mathematical result and prints it to the screen.

Tip: read the instruction with
`scanf("%f%c%f%*c",&getal1,&operand,&getal2);`
and use `switch(operand)`

```
Enter an expression (without spaces!): 245/16
245 / 16 = 15.31
```

**5.2.14.** A gas company calculates its prices as follows:

- for a consumption ≤ 1m³ € 20,00
- for a consumption > 1m³ € 20 for the first m³ + € 3,5/m³ for the part over 1m³

The total amount needs to be increased with a tax of 21%.
Write a program that asks the user to enter his/her consumption and prints an invoice to the screen showing consumption, cost prince, tax amount and total amount.

**5.2.15.** The equivalent resistance ($R_{eq}$) for 2 parallel resistors $R_1$ and $R_2$ can be calculated with:

$$\frac{1}{R_{eq}} = \frac{1}{R_1} + \frac{1}{R_2}$$

Write a program that reads the values of $R_1$ and $R_2$, calculates $R_{eq}$ and prints it to the screen. Make sure the program also works correctly for an $R_1$ and/or $R_2$ equal to 0!

**5.2.16.** Write a program that calculates the length of the third side of a right-angled triangle based upon the lengths of the 2 other sides and prints it to the screen. To this end, ask the user to enter the length of all 3 sides in the order: $side_1$ $side_2$ hypotenuse. A zero is to be entered for the side for which you wish to calculate the length.

$$(hypotenuse)^2 = side_1^2 + side_2^2$$

**5.2.17.** Write a program that reads 3 integer numbers that are to be interpreted as a date and prints the corresponding day of the week.

The day of the week can be calculated as follows:

$$day\ of\ the\ week\ =\ factor - \left[\frac{factor}{7}\right] * 7$$

where

- [X] equals the integer part of X
- day of the week is represented by 0 to 6 for Saturday till Friday.
- $factor = 365 * year + day + 31 * (month - 1) + \frac{year-1}{4} - \left[\left(\frac{year-1}{100} + 1\right) * \frac{3}{4}\right]$

  $valid\ for\ January\ and\ Februari$
- $factor = 365 * year + day + 31 * (month - 1) - (0.4 * month + 2.3) + \frac{year}{4} - \left[\left(\frac{year}{100} + 1\right) * \frac{3}{4}\right]$

  $valid\ for\ March\ till\ December$

**5.2.18.** Write a program that asks the user to enter the coefficients a, b and c of a quadratic equation with general form $ax^2 + bx + c = 0$ and prints all real roots of this equation. Make sure the program does not crash for certain values of the coefficients!

| **a** | **b** | **c** | **roots** |
|---|---|---|---|
| 1 | 1 | -6 | two real roots: -3 and 2 |
| 2 | 1 | -6 | two real roots: -2 and 1.5 |
| 1 | 1 | 1 | no real roots |
| 1 | 4 | 4 | double real root: 2 |
| 0 | 1 | 1 | one solution: -1 |
| 0 | 0 | 1 | no solutions (inconsistent equation) |
| 0 | 0 | 0 | identity |

## 5.3 Repetition statements

In every programming language there are circumstances were you want to do the same thing many times. For instance you want to print the same words ten times. You could type ten `printf` functions, but it is easier to use a loop. To this end, C provides 3 repetition statements:

- `for` statement
- `while` statement
- `do … while` statement

### 5.3.1 The `for` statement

The `for` statement is used for counter-controlled repetition. It allows to **iterate** a block of statements a **predefined number** of times.

In its most general form, the `for` loop can be described as:

```
for(<initialization>; <test expression>; <step expression>)
{
    <statement>;
    <statement>;
}
<next statement>;
```

A counter variable is used to count the number of repetitions. At the start of the `for` loop, the counter variable needs to be initialized once as expressed in the `<initialization>` expression. As long as the `<test expression>` is met (result ≠ 0), the block statements are executed and the counter variable is increased / decreased. The `<step expression>` describes how the counter variable needs to be changed **after** every loop execution. When the counter variable no longer matches the `<test expression>` (the test yields 0), the loop is terminated and execution continues with the `<next statement>` after the `for` loop.
In a flowchart, the `for` loop looks like:

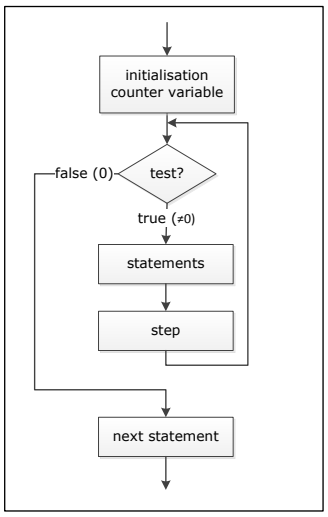



**Figure 18: flowchart `for` loop**

**Learning note**

Note that the braces in the `for` loop are always placed right under the first letter of `for` and that all body statements are indented! This improves the readability of the code!

Example:
read 10 integer numbers and print their sum.

Figure 19 shows the flowchart of an algorithm using a `for` loop.

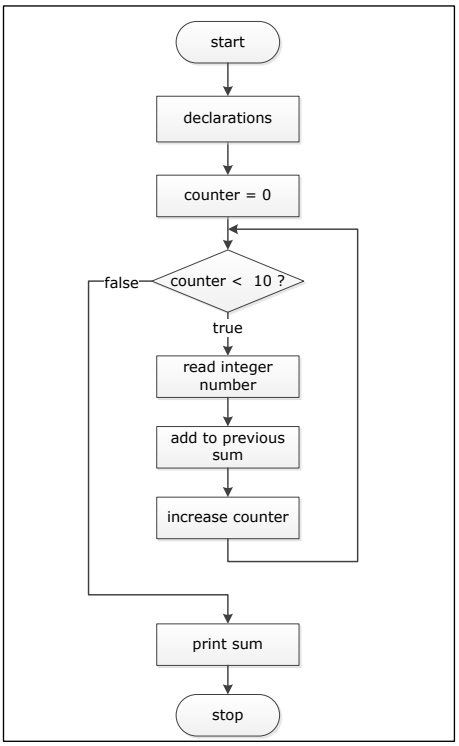
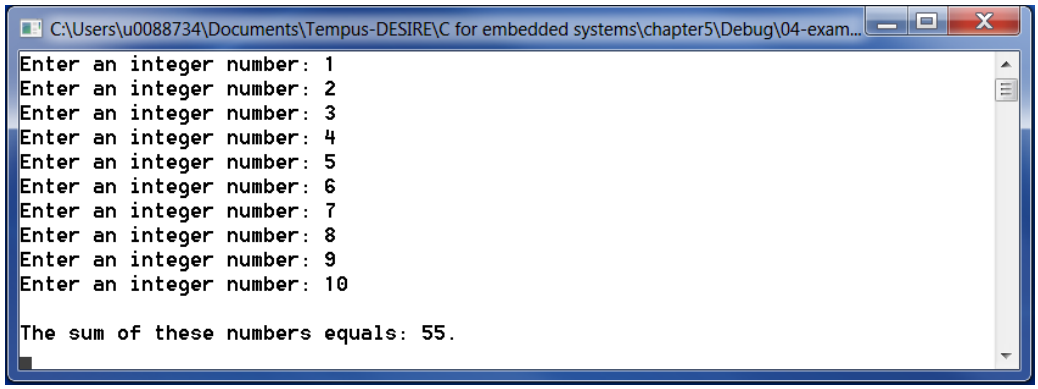



**Figure 19: flowchart `for` loop example**

The corresponding C code and screen output are shown in Code 13.

```
1   # include <stdio.h>
2
3   # define REP_TIMES 10
4
5   int main(void)
6   {
7      int number, counter, sum=0;
8
9      for (counter = 0; counter < REP_TIMES; counter++)
10     {
11            printf("Enter an integer number: ");
12            scanf("%d%*c", &number);
13            sum += number;
14     }
15
16     printf("\nThe sum of these numbers equals: %d.\n", sum);
17
18     return 0;
19  }
```




**Code 13: for loop example**

**Remark**

If only one statement is to be repeated, the braces can be left out:

```
for(i=0; i< line_length; i++)
      printf("-");
printf("\n");
```

**Common mistake**

The counter is changed by the step expression written at the start of the for loop. As such, the counter does not need to be changed inside the loop!

### 5.3.2 The `while` statement

The `while` statement is used for condition-controlled repetition. It allows to iterate a block of statements as long as a condition remains true. In this case it is usually **not know** in advance **how many times** the loop will be executed. Of course, the loop must be written such that the condition will become false at some point.

In its most general form, the `while` loop can be described as:

```
<initialization>;
while (<condition>)
{
    <statement>;
    <statement>;
    …
    <change condition variables>;
}
<next statement>;
```

Some condition variables are used to determine whether the loop needs to be executed again or not. These condition variables need to be initialized before the start of the `while` loop in the `<initialization>` statement. As long as the `<condition>` is true (result ≠ 0), the block statements are

executed. To avoid ending up with an infinite loop, at least one of these loop statements needs to take care of changing the condition variables such that in the end the condition will become false (0). When this happens, the loop is terminated and execution continues with the `<next statement>` after the `while` loop.

In a flowchart, the `while` loop looks like:

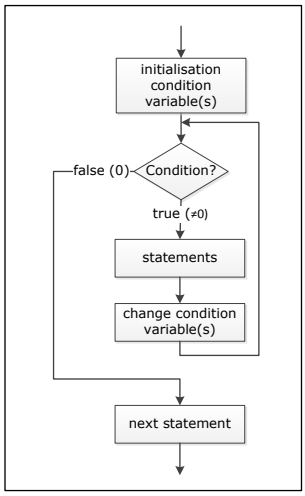



**Figure 20: flowchart of `while` loop**

**Learning note**

Note that the braces in the `while` loop are always placed right under the first letter of `while` and that all body statements are indented! This improves the readability of the code!

Example:
read integer numbers until the number 999 is entered and print their sum.

Figure 21 shows the flowchart of an algorithm based on a `while` loop.

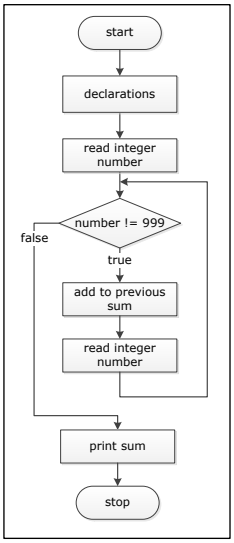
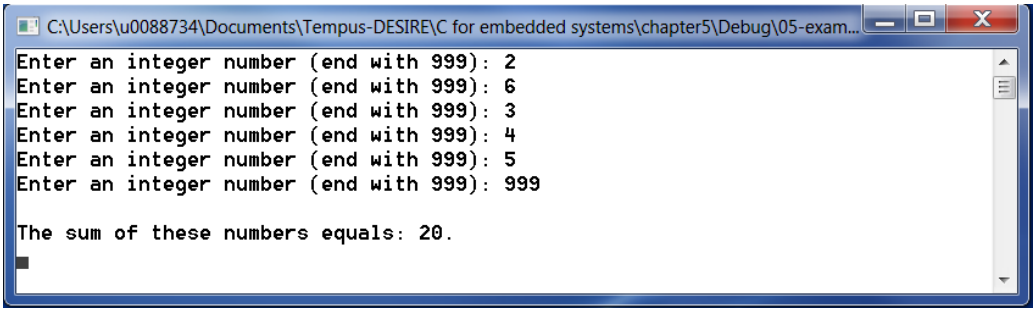



**Figure 21: flow chart `while` loop example**

The corresponding C code and screen output are shown in Code 14:

```
1   /*
2      read integer numbers until the number 999 is entered and print
       their sum
3   */
4   # include <stdio.h>
5
6   int main(void)
7   {
8      int number, sum;
9
10     sum = 0;
11     printf("Enter an integer number (end with 999): ");
12     scanf("%d%*c", &number);
13
14     while (number != 999)
15     {
16            sum += number;
17            printf("Enter an integer number (end with 999): ");
18            scanf("%d%*c", &number);
19     }
20
21     printf("\nThe sum of these numbers equals: %d.\n", sum);
22
23     return 0;
24  }
```




**Code 14: while loop example**

**Common mistake**

The condition variables need to be changed explicitly inside the `while` loop! This is often forgotten, resulting in an endless loop!

**Common mistake**

The condition variables need to be initialized correctly before entering the loop, otherwise the results of the program will probably be incorrect!

### 5.3.3 The `do … while` statement

The `do … while` repetition statement is similar to the `while` repetition statement.

In its most general form, the `do … while` loop can be described as:

```
<initialization>;
do
{
    <statement>;
    <statement>;
    …
    <change condition variables>;
}while(<condition>);
<next statement>;
```

In the `do … wile` statement, the condition is tested **after** the execution of the loop statements, whereas in the `while` statement, the condition is tested first. As a result, the loop statements in a `do … while` loop are always executed at least once. In a `while` loop on the other hand, it is possible that the loop statements are never executed.

**Learning note**

Note that the braces in the `do … while` loop are always placed right under the first letter of `do` and that all body statements are indented! This improves the readability of the code!

Example 1:
read integer numbers until the number 999 is entered and print their sum.

Figure 22 shows the flowchart of an algorithm based on a `do … while` loop.

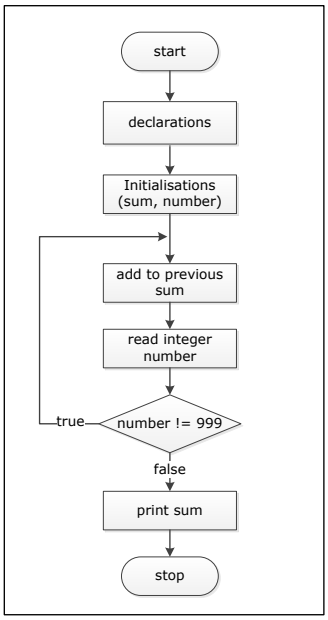
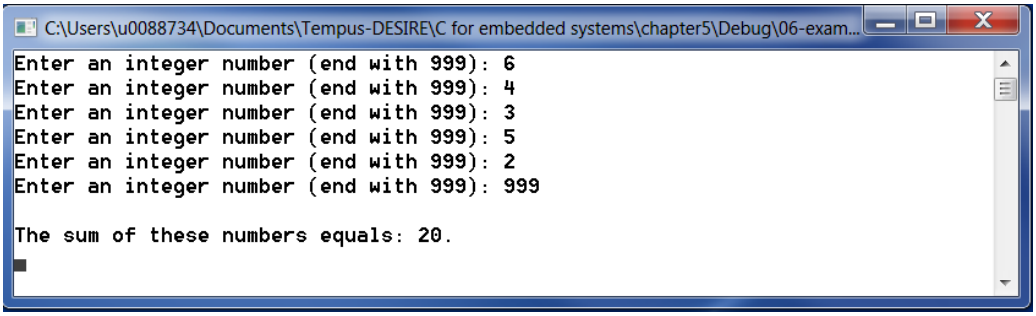



**Figure 22: flow chart do ... while example 1**

The corresponding C code and screen output are shown in Code 15:

```
1   /*
2      read integer numbers until the number 999 is entered and print
       their sum
3   */
4   # include <stdio.h>
5
6   int main(void)
7   {
8      int number, sum;
9
10     sum = 0;
11     number = 0;
12
13     do
14     {
15            sum += number;
16            printf("Enter an integer number (end with 999): ");
17            scanf("%d%*c", &number);
18     } while (number != 999);
19
20     printf("\nThe sum of these numbers equals: %d.\n", sum);
21
22     return 0;
23  }
```




**Code 15: `do ... while` example 1**

**Common mistake**

The semicolon at the end of the `do … while` is often forgotten!

Example 2:
ask the user to choose between the symbols '1', '2', '3' and '4'. Keep asking for a new choice until a valid symbol was entered.

This can be achieved by following code:

```
1   # include <stdio.h>
2   int main(void)
3   {
4      char choice;
5
6      do
7      {
8             printf("Enter your choice [1, 2, 3 of 4]: ");
9             scanf("%c%*c", &choice);
10     } while (choice < '1' || choice > '4');
```

```
11
12      printf("Your choice: %c", choice);
13
14      return 0;
15  }
```

**Code 16: `do ... while` example 2**

## 5.3.4 `break` and `continue`

`break` and `continue` statements are used to change the program flow in loops.

### 5.3.4.1 `break`

In section 5.2.3 we discussed the use of `break` in a `switch` statement. There the `break` was used to immediately stop the execution of the `switch` and continue with the next statement after the `switch`.

A `break` statement can also be used in a `for`, `while` or `do … while` loop. In these cases, the `break` statement will immediately terminate the execution of the nearest enclosing loop and will pass control to the statement following that loop.

Using a `break` statement in a `for` or `while` loop can be represented by the flowchart in Figure 23.

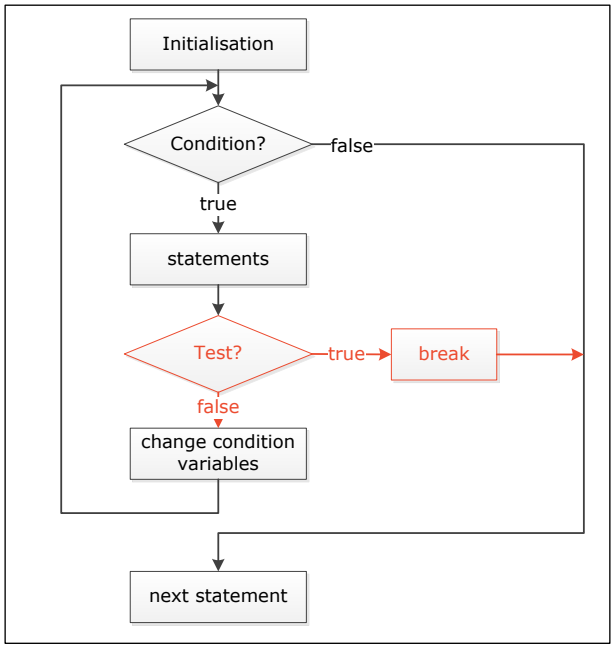



**Figure 23: flowchart `break`**

Example:
read integer numbers until the number 999 is entered and print their sum. If the sum exceeds 50, the loop must be terminated as well.

A possible solution is to use a `break` statement inside the loop:

```
1   /*
2      Read integer numbers until the number 999 is entered and print
3     their sum. If the sum exceeds 50, the program must be terminated.
4   */
5   # include <stdio.h>
6
7   int main(void)
8   {
9      int number, sum;
10
11     sum = 0;
12     printf("Enter an integer number (end with 999): ");
13     scanf("%d%*c", &number);
14
15     while (number != 999)
16     {
17            sum += number;
18            if (sum > 50) break;
19            printf("Enter an integer number (end with 999): ");
20            scanf("%d%*c", &number);
21     }
22
23     printf("\nThe sum of these numbers equals: %d.\n", sum);
24
25     return 0;
26  }
```

**Code 17: `break` example**

A better solution in this case would be to check the sum in the condition of the `while` loop. The C code for the loop then looks like:

```
15  while (number != 999 && sum <= 50)
16     {
17            sum += number;
18            printf("Enter an integer number (end with 999): ");
19            scanf("%d%*c", &number);
20     }
```

### 5.3.4.2 `continue`

When a `continue` statement is used in a `for`, `while` or `do … while` loop, the remainder of the loop statements is skipped and the next loop iteration is performed.

In the case of a `while` or `do … while` control structure, the loop condition is tested immediately after the execution of the continue statement. In a `for` repetition statement on the other hand, the step expression is executed first before evaluating the loop condition.

**Common mistake**

Changing the loop condition variables of a `while` or `do … while` control structure **after** the `continue` statement can result in an endless loop!

The `continue` statement can be included in a flowchart as shown below:

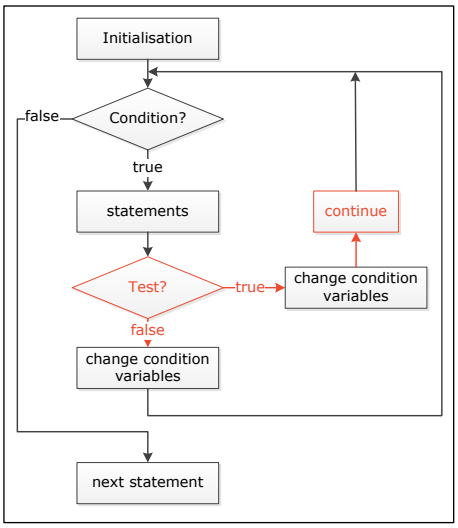



**Figure 24: continue in `while` loop**

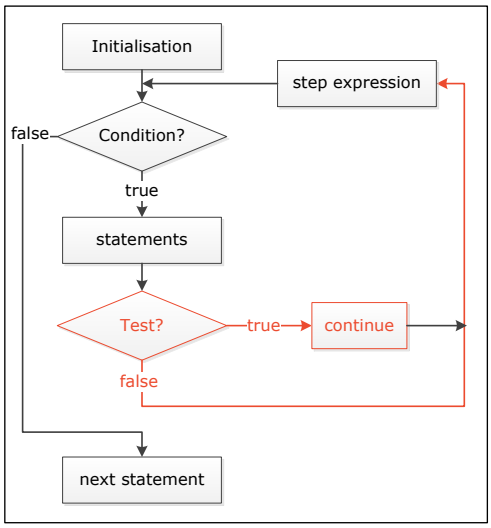



**Figure 25: continue in `for` loop**

Example:
read integer numbers until the number 999 is entered and print the sum of all positive numbers read.

A possible solution using a `continue` statement is described in Code 18:

```
1   /*
2      read integer numbers until the number 999 is entered and print
3      the sum of all positive numbers
4   */
5   # include <stdio.h>
6
7   int main(void)
8   {
9      int number, sum;
10
11     sum = 0;
12
13     printf("Enter an integer number (end with 999): ");
14     scanf("%d%*c", &number);
15
16     while (number != 999)
17     {
18            if (number < 0)
19            {
20                   printf("Enter an integer number (end with 999): ");
21                   scanf("%d%*c", &number);
22                   continue;
23            }
24            sum += number;
25            printf("Enter an integer number (end with 999): ");
26            scanf("%d%*c", &number);
27     }
```

```
28
29     printf("\nThe sum of these numbers equals: %d.\n", sum);
30
31     return 0;
32  }
```

**Code 18: `continue` statement**

Also in this case, the `continue` statement is not really necessary. The loop statements could also be rewritten as follows:

```
16     while (number != 999)
17     {
18            if (number >= 0)
19                   sum += number;
20            printf("Enter an integer number (end with 999): ");
21            scanf("%d%*c", &number);
22     }
```

**Learning note**

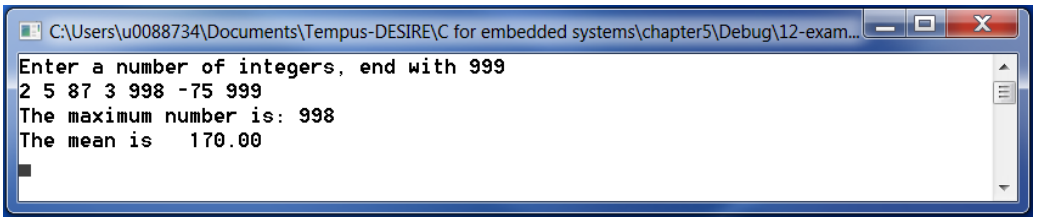
Avoid using `break` and `continue` statements in loops!

## 5.3.5 Loop examples

### 5.3.5.1 Example 1

Write a C program that reads a number of integers and prints their mean and maximum value. To end the reading process, the user enters 999. Make sure 999 is not taken into account in your calculations.

```
1   #include <stdio.h>
2   int main(void)
3   {
4      int number, sum, counter, max;
5      float mean;
6      sum = counter = 0;
7      printf("Enter a number of integers, end with 999 \n");
8      scanf("%d%*c", &number);
9      while (number != 999)
10     {
11            sum += number;
12            counter++;
13            if (counter == 1 || number > max) max = number;
14            scanf("%d%*c", &number);
15     }
16     if (counter == 0)
17            printf("No integers were entered! \n");
18     else
19     {
20            printf("The maximum number is: %d \n", max);
21            mean = (float)sum / counter;
22            printf("The mean is %8.2f \n", mean);
23     }
24     return 0;
25  }
```



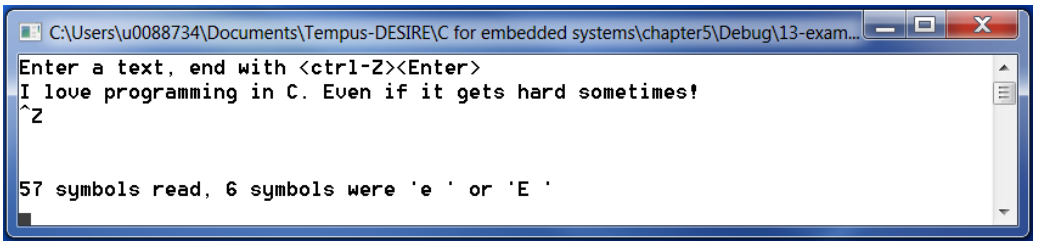

**Code 19: loop example 1**

### 5.3.5.2 Example 2

Write a C program that reads a text symbol by symbol and prints the number of symbols read and how many times the letter 'e' or 'E' occurs in the text. To end the reading process, the user enters <Ctrl-Z><Enter> (`EOF`).

To solve this problem, we will make use of the function `getchar()` that reads 1 symbol and returns the ASCII value of that symbol. If <Ctrl-Z><Enter> is entered, `getchar` will return `EOF` (defined as -1 in `stdio.h`).

```
1   #include <stdio.h>
2   int main(void)
3   {
4      char c;
5      int number_symbols = 0;
6      int number_e = 0;
7      printf("Enter a text, end with <ctrl-Z><Enter> \n");
8      c = getchar();
9      while (c != EOF)
10     {
11            number_symbols++;
12            if (c == 'e' || c == 'E') number_e++;
13            c = getchar();
14     }
15     printf(" \n\n");
16     printf("%d symbols read, ", number_symbols);
17     printf("%d symbols were \'e \' or \'E \'\n", number_e);
18     return 0;
19  }
```




**Code 20: loop example 2**

### 5.3.6 Exercises

**5.3.1.** Write a program with only 1 variable, that prints the numbers -3, -1, 1, 3, 5, …, 25, comma separated to the screen. Nothing needs to be read from the keyboard.

**5.3.2.** Write a program that asks the user to enter an integer number and prints the multiplication table of that number

The screen dialog should look like:

```
Enter an integer number: 7
The table of multiplication of 7 is:
1 x 7 = 7
2 x 7 = 14
...
20 x 7 =140
```

**5.3.3.** Write a program that asks the user to enter an integer number n and prints the sum 1+2+3+4+…+n.

**5.3.4.** Write a program that prints a filled square to the screen by printing 22 lines of 40 black rectangles (the ASCII code for a black rectangle is 219).

**5.3.5.** Write a program that first asks the user to enter an integer number. Afterwards more integers are asked until the sum of those integers equals or exceeds the first number entered.

```
Enter the limit: 15
Enter an integer number: 3
Enter an integer number: 4
Enter an integer number: 6
Enter an integer number: 5
The limit of 15 is reached or exceeded!
```

**5.3.6.** Write a program that asks the user to enter an integer number in the interval [-2, 4.5[ (i.e. $-2 \leq number < 4.5$) and prints it to the screen. If a wrong number is entered, a new number must be requested until a valid number is entered.

**5.3.7.** Write a program that asks the user to enter an integer number in the interval [-30 , 30]. Make sure only valid numbers can be entered! This number is then printed as a bar graph made out of * symbols. Negative numbers are drawn from the middle to the left, positive numbers are drawn from the middle to the right. Points are used to fill up the empty places. The entered number itself is printed in the end.
Repeat the program until a 0 is entered.

```
number: 12
.............................|***********.................
number: -4
........................****|.............................
number: 0
```

**5.3.8.** Write a program that sums 10 numbers entered by the user and computes the mean of those 10 numbers. Try to use only 3 variables in your program. Calculate the sum while reading the numbers.

**5.3.9.** Write a program that prints the mean of a number of integers. The exact number of integers is not known upfront. If the number 999 is read, the program stops reading new numbers. 999 cannot be taken into account for the calculation of the mean.

**5.3.10.** Write a program that reads a natural number n, calculates n! and prints the result to the screen.
0! = 1
n! = 1 x 2 x 3 x … x n with n > 0

**5.3.11.** Write a program that reads 2 numbers: a base b ($b \in \mathrm{R}$) and an exponent n ($n \in N$). Afterwards, the exponentiation $b^n$ is calculated and the result is printed to the screen. Calculate $b^n$ with a loop. Do not use a standard function.

**5.3.12.** Repeat exercise 5.3.11 but with n being an integer (both positive and negative values are possible).

**5.3.13.** Write a program that prints the minimum and maximum value of 10 numbers entered by the user. Use only 4 variables in your program.

**5.3.14.** Repeat exercise 5.3.13 but this time also print when the minimal and maximal number were entered.

```
Enter 10 numbers: 5 98 6 -5 78 -20 4 6 8 2
maximum: 98 at place 2
minimum: -20 at place 6
```

**5.3.15.** Write a program that calculates the greatest common divisor of 2 positive integers. Use Euclid's algorithm to determine the gcd. In this algorithm, the biggest number is replaced by the difference of both numbers. Repeat this until both numbers are equal. This number is the gcd. Print the gcd and all intermediate steps.

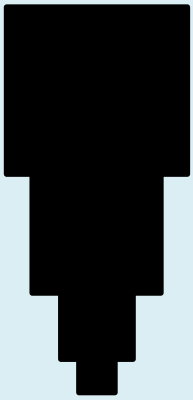
```
Enter 2 positive integer numbers: 114 90

114      90
 24      90
 24      66
 24      42
 24      18
  6      18
  6      12
  6      6

The gcd of 114 and 90 equals 6.
```

**5.3.16.** Write a program that draws a tower upside down using the ASCII character 219 (█). The tower consists of a sequence of squares. The top square has a side defined by the user. The next square is 2 blocks smaller, the next one is again 2 blocks smaller, … New towers are drawn until the user enters 'n'.

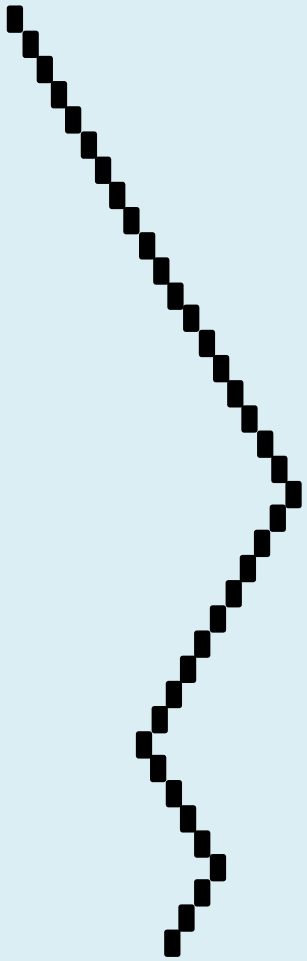
```
This program builds a tower upside down!
How wide should the tower be?
(enter an odd number between 3 and 15)
7
```



```
Do you want to build another tower? (y/n) y
How wide should the tower be?
…
```

**5.3.17.** Write a program that draws a zigzag line until the user enters 'n' to stop. The first portion of the line is made out of a number of blocks determined by the user. The next portion is only half that length, … If the user wants to draw several zigzag lines, make sure they are all well positioned and drawn correctly.

```
Welcome to ZIGZAG world!
How wide do you want your ZIGZAG line?
Enter a number between 3 and 60: 20
```



```
Do you want to draw another ZIGZAG line? (y/n) n
```

**5.3.18.** Write a program that determines the maximum out of 10 numbers entered by the user. The program also prints the number of times the maximal value was entered.

**5.3.19.** Write a program that asks the user to enter a positive integer and prints all rows of consecutive positive integers with a sum equal to the first integer entered.

```
Enter an integer > 0: 87
87 is the sum of:
12 13 14 15 16 17
28 29 30
43 44
```

```
Enter an integer > 0: 64
64 is the sum of:
no solution
```

# 6 Functions

**Objectives**

In this chapter you will learn to write more complex programs using functions. Both, the built-in standard functions as well as programmer-defined functions, will be discussed.
As a special case, the generation of random numbers will be treated.
You will learn how to pass information to and retrieve information from a function. Attention will also be given to the scope of variables used in the different functions of your program.

Most computer programs are built to solve more complex problems than the ones we have treated so far. To develop a computer program for complex problems, it is good practice to divide the problem into smaller and more manageable sub-problems or modules. Then, for each of those modules, an algorithm must be written and translated into C code.

To support this process of structured programming, C provides the concept of functions. At least one function is always present in every C program: the `main()` function! A function in C is a block of code that performs a task and then returns control to a caller. It has its own name and can be called from different places in the program. As such, functions can also be used to avoid repetition of code.

C provides 2 types of functions: standard functions and programmer-defined functions. Both types of functions will be discussed in this chapter.

## 6.1 Standard functions

The C standard library provides a lot of functions that perform common mathematical calculations, string manipulations, input/output, and many other useful tasks. In this section, we will discuss only a few of these libraries. It is worthwhile to look into the full set of standard functions provided.

Like everything in C, functions need to be declared before they can be used in a program. The declarations of standard functions are grouped into header files (one header file per standard library). This is why we have included "`<stdio.h>`" in all previous programs. This header file contains the declarations of functions like `printf()`, `scanf()`, …

The function definitions are inserted by the linker that will look for the correct definitions in the corresponding library. For the functions `printf()`, `scanf()` for instance, this will be the `stdio` library.

### 6.1.1 Mathematical standard functions

The mathematical standard functions supported by C are grouped into the standard library "`math`".

To include the necessary function declarations, the header file "`math.h`" needs to be included in the program.

Table 8 shows an overview of the most commonly used math functions provided:

| Function declaration | description | example |
|---|---|---|
| `double cos(double x);` | $\cos(x)$ $(x\ in\ radians)$ | $\cos\left(\frac{\pi}{2}\right) = 0.0$ |
| `double sin(double x);` | $\sin(x)$ $(x\ in\ radians)$ | $\sin\left(\frac{\pi}{2}\right) = 1.0$ |
| `double tan(double x);` | $\tan(x)$ $(x\ in\ radians)$ | $\tan\left(\frac{\pi}{4}\right) = 1$ |
| `double acos(double x);` | $\arccos(x)$ $(res\ in\ radians)$ | $\text{acos}(1.0) = 0.0$ |
| `double asin(double x);` | $\arcsin(x)$ $(res\ in\ radians)$ | $\text{asin}(0.0) = 0.0$ |
| `double atan(double x);` | $\arctan(x)$ $(res\ in\ radians)$ | $\text{atan}(0.0) = 0.0$ |
| `double atan2(double y, double x);` | $\arctan\left(\frac{y}{x}\right)$ $(res\ in\ radians)$ | $\text{atan}(1{,}4) = 0.24$ |
| `double exp(double x);` | $e^x$ | $\exp(1.0) = 2.718$ |
| `double log(double x);` | $\ln(x)$ | $\log(2.718) = 1.0$ |
| `double log10(double x);` | $\log(x)$ | $\log(1000) = 3$ |
| `double pow(double x, double y);` | $x^y$ | $\text{pow}(2{,}5) = 32.0$ |
| `double sqrt(double x);` | $\sqrt{x}$ | $\text{sqrt}(9.0) = 3.0$ |
| `double floor(double x);` | rounds x to the largest integer ≤ x | $\text{floor}(123.54) = 123.0$ |
| `double ceil(double x);` | rounds x to the smallest integer ≥ x | $\text{ceil}(123.54) = 124.0$ |
| `double fabs(double x);` | absolute value of x ($\lvert x \rvert$) | $\text{fabs}(-2.5) = 2.5$ |

**Table 8: math library functions**

**Learning note**

The function `pow` uses calculation methods that need a lot of computations. Therefore, do not use the function `pow` for simple exponentiations like $x^2$ and $x^3$. Use $x * x$ or $x * x * x$ instead.

Example:
Write a program that prints a table with the cosine of all angles between 0° and 360°. Change the angles with steps of 30°.

This problem can be solved as indicated by the flowchart of Figure 26:

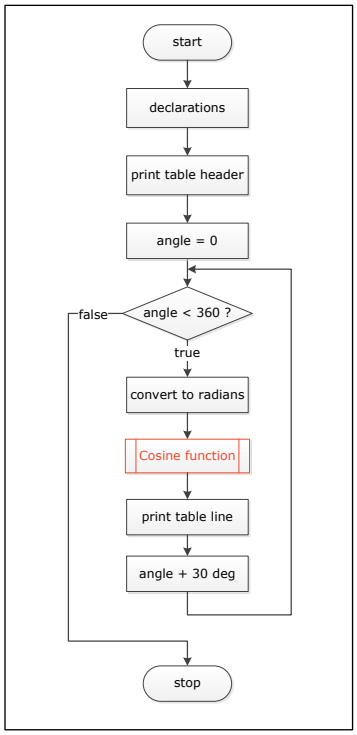
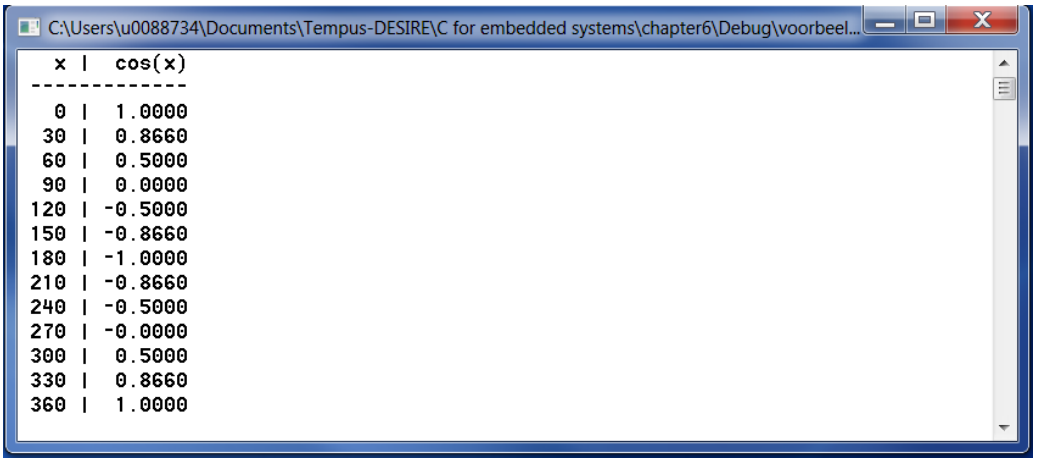



**Figure 26: flowchart standard function cos example**

The corresponding C code and screen output are shown in Code 21.

```
1   /*
2      this program prints the cosine of angles between 0 and 360
3     degrees
4   */
5   #include <stdio.h>
6   #include <math.h>
7   #define M_PI 3.14159265358979323846
8
9   int main(void)
10  {
11     int degrees;
12     double radians, res;
13
14     printf(" x | cos(x) \n");
15     printf(" ----------------- \n");
16
17     for (degrees = 0; degrees <= 360; degrees += 30)
18     {
19            radians = (double)degrees*M_PI / 180;
20            res = cos(radians);
21            printf(" %3d | %7.4lf \n", degrees, res);
22     }
23
24     return 0;
25  }
```




**Code 21: example math standard library**

The line "`#include <math.h>`" takes care of including all math function declarations into the program. As a result, the cosine function can now be used.

To use a function, simply write the name of that function followed by parenthesis (). If the function used requires one or more inputs (=arguments), these arguments need to be placed in between the parenthesis in the order specified by the function declaration.

With the line "`res=cos(radians);`", the math standard function "`cos`" is called to compute the cosine of the angle "`radians`". Remark that `radians` is a variable of the type double. This is needed since the cosine function takes a double as argument as can be seen in the function declaration. The assignment operator "`=`" takes care of assigning the result of the cosine calculation to the variable `res`. Also this variable needs to be of the type double as dictated by the function declaration.

The line "`#define M_PI 3.14159265358979323846`" defines `M_PI` to be equal to 3.14159265358979323846. The precompiler will replace every `M_PI` in the program by this number. Commonly used mathematical constants are also defined in "`math.h`" To use them add "`#define _USE_MATH_DEFINES`" before the line "`#include <math.h>`". The explicit definition of `M_PI` in the program is now no longer needed.

## 6.1.2 Other standard functions

In this section an overview of the most commonly used functions of some standard libraries will be given.

### 6.1.2.1 Functions for string manipulations `<string.h>`

```
int strlen(char *s);
int strcmp(char *s, char *t);
int strncmp(char *s, char *t, int n);
char *strcpy(char *s, char *t);
char *strncpy(char *s, char *t, int n);
```

```
char *strcat(char *s, char *t);
char *strncat(char *s, char *t, int n);
char *strchr(char *s, int ch);
```

#### 6.1.2.2 Functions for character handling <ctype.h>

```
int isalnum(int ch);
int isalpha(int ch);
int iscntrl(int ch);
int isdigit(int ch);
int isgraph(int ch);
int islower(int ch);
int isprint(int ch);
int ispunct(int ch);
int isspace(int ch);
int isupper(int ch);
int isxdigit(int ch);
int tolower(int ch);
int toupper(int ch);
```

#### 6.1.2.3 Standard lib functions <stdlib.h>

```
double atof(char *s);
int atoi(char *s);
long atol(char *s);
exit(int status);
int system(char *s);
int abs(int n);
long labs(long n);
int rand(void);
```

#### 6.1.2.4 More C libraries

| | |
|---|---|
| `<errno.h>` | error handling |
| `<float.h>` | floating point precision |
| `<signal.h>` | signals |
| `<stddef.h>` | special types |
| `<time.h>` | dates and time |

### 6.1.3 Generation of random numbers

To generate a random number, the C standard library function `rand()`, declared in the header file `stdlib.h`, can be used. This `rand()` function generates a random integer between 0 and `RAND_MAX` (a constant defined in `stdlib.h`).

To demonstrate the `rand()` function, let's write a program that prints 10 random integers ranging from 1 to 100.

Simply using the function `rand()` results in integers that can be much larger than 100! To limit the numbers to 100, the `rand()` function output needs to be scaled down. This can be accomplished by computing the remainder of the integer division by 100:

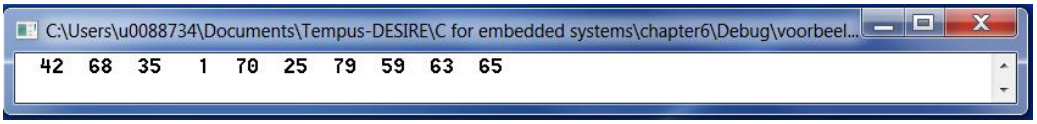
```
rand()%100
```

resulting in an integer number ranging from 0 to 99. Shifting this result to the interval [1, 100] can then be done by adding 1:

```
rand()%100 + 1
```

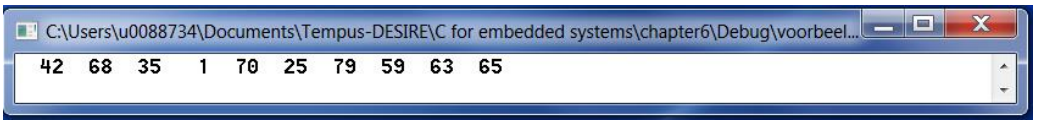
The program that prints 10 of these random numbers can then be written as shown in Code 22:

```
1   #include <stdio.h>
2   #include <stdlib.h> //contains definition of rand()
3
4   int main(void)
5   {
6      int i;
7      for (i = 0; i<10; i++)
8             printf("%4ld", rand() % 100 + 1);
9      printf(" \n");
10     return 0;
11  }
```




**Code 22: random number generation**

Executing the program of Code 22 again produces:




So it appears that these numbers are not that random after all! Actually, the function `rand()` generates pseudo-random numbers. Calling the `rand` function repeatedly produces a series of numbers that appears to be random. However, this sequence is repeated each time the program is executed. This pseudo-random behavior is very useful during the debug stage of a program. It is needed to reproduce and effectively solve possible programming errors.

However, once a program is debugged, it should be possible to generate true random numbers. To this end, the function `srand()` can be used. This function takes an unsigned integer as input and determines a starting value or `seed` for the `rand()` function. Different values for the `seed` parameter, will result in different series of random numbers. As such, to obtain different results each time the program is executed, the seed needs to be changed in every run. This can be done automatically be using the function `time(NULL)`, that will return the number of seconds elapsed since 0 o'clock January 1st 1970. The function `time()` is declared in the header file `time.h`.

Including the function `srand()` in the example of Code 22 results in:

```
1   #include <stdio.h>
2   #include <stdlib.h>    //needed for the functions rand() and srand()
3   #include <time.h>      //needed for the function time()
4
5   int main(void)
6   {
7      int i;
8      srand(time(NULL));
9      for (i = 0; i<10; i++)
10           printf("%4d", rand() % 100 + 1);
11     printf(" \n");
12     return 0;
13  }
```

**Code 23: usage of `srand()` function**

Executing this program several times results in different screen outputs as can be seen in Figure 27.

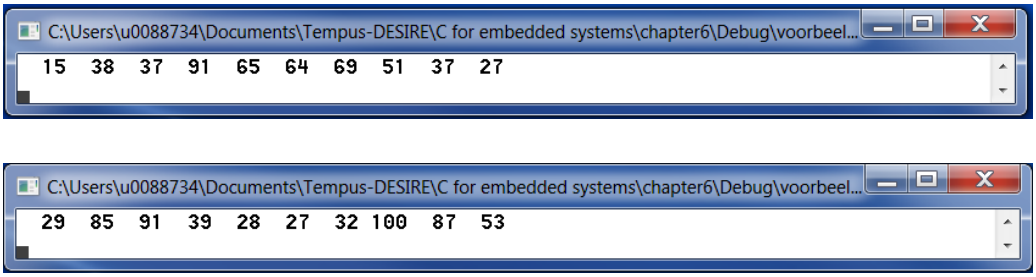



**Figure 27: random number generation using `srand()`**

Example:
Write a program that picks a random number in the interval [1,100]. The user needs to guess the number chosen. If the guess was too high or too low, the program prints "too high" or "too low" respectively. If the guess is correct, the program prints the number of guesses needed to find the secret number.

```
1   /*
2      game: guess the secret number
3   */
4   #include <stdio.h>
5   #include <stdlib.h>
6   #include <time.h>
7
8   int main(void)
9   {
10     int secret, number;
11     int counter;
12
13     counter = 0;
14     srand(time(NULL));
15     secret = rand() % 100 + 1;
16     printf("Search the secret number in the interval [1,100] \n");
```

```
17      do
18      {
19              printf("Try your luck: ");
20              scanf("%d%*c", &number);
21              counter++;
22              if (number > secret) printf("Your guess was too high!\n");
23              if (number < secret) printf("Your guess was too low!\n");
24      } while (number != secret);
25
26      printf("\nYou needed %d guesses to find the secret number!\n",
27                      counter);
28      return 0;
29  }
```

**Code 24: guess the secret number**

## 6.2 Programmer-defined functions

Next to the provided standard functions, programmers can use self-defined functions to accomplish specific tasks that were not included in the standard libraries. Using self-defined functions is needed to split the problem in sub problems, to avoid code repetition and to be able to reuse code in different programs.

Every function can be described as a black box, that takes inputs or arguments and has a certain result or return value. This function principle is shown in Figure 28.


**Figure 28: general function principle**

Like standard functions, also custom functions need a declaration and a definition. The declaration needs to be written before the first function call and contains the function name and the types of the return value and parameters or inputs of the function:

```
return-value-type <function_name>(list of parameter types);
```

The function definition can be written anywhere in the code. The format of a function definition is:

```
return-value-type <function_name>(list of parameters + their types)
{
    statements;
}
```

**Learning note**

To avoid compile errors and to keep the code well-structured always adhere to following style rules:

- write the function declarations in the beginning of the source code, before the main function definition.
- write the custom function definitions after the main function definition.

Once the function declaration and definition are written, the function can be called from everywhere in the program. Like standard functions, custom functions are called using the function name followed by parenthesis. If arguments are to be passed on to the function, these arguments are put in between the parenthesis.

A flowchart showing the program flow when a function is used is shown in Figure 29:

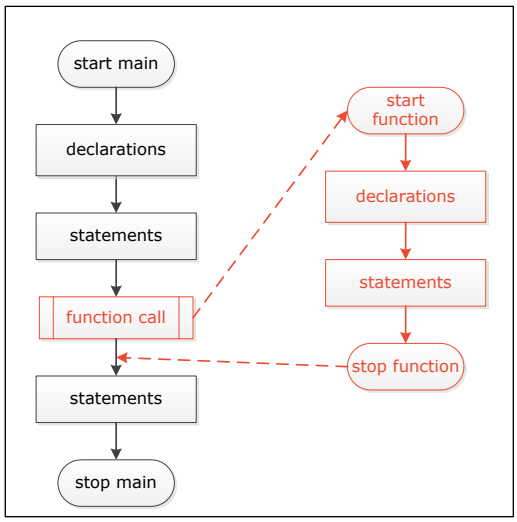



**Figure 29: function flowchart**

To increase the readability of the flowcharts, the dotted lines will be left out from now on!

## 6.2.1 Void functions without parameters

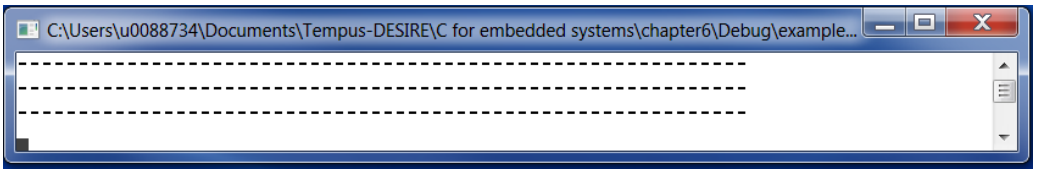
Void functions are functions without return value. In its most general form, a void function without parameters can be described as:

```
void <function_name>(void);
```

To indicate that there is no return value, the function name is preceded by the word `void`. Simply omitting a return-value-type will result in the assumption of an integer return value. As such, the word `void` before the function name is absolutely necessary to indicate that no return value will be present. In the same way, the word `void` is used in between the parenthesis following the function name to indicate that no parameters are needed.

Example:
Write a program that prints several lines of 60 '-' characters.

```
1   #include <stdio.h>
2   #define LINELENGTH 60
3
4   void line(void);        /* function declaration */
5
6   int main(void)
7   {
8      line();              /* function call */
9      line();
10     line();
11     return 0;
12  }
13
14  void line(void)         /* function definition */
15  {
16     int i;
17
18     for(i=0; i<LINELENGTH; i++)
19     {
20        printf("-");
21     }
22
23     printf("\n");
24  }
```




**Code 25: example void functions without parameters**

## 6.2.2 Void functions with parameters

In many cases, the caller needs to pass information to the function. This is done by the use of function parameters that are written between parenthesis after the function name. These parameters can then be used as local variables inside the function. In the function definition, every parameter is given a name and type resulting in:

```
void <function_name>(type par1, type par2, …)
{
       statements;
}
```

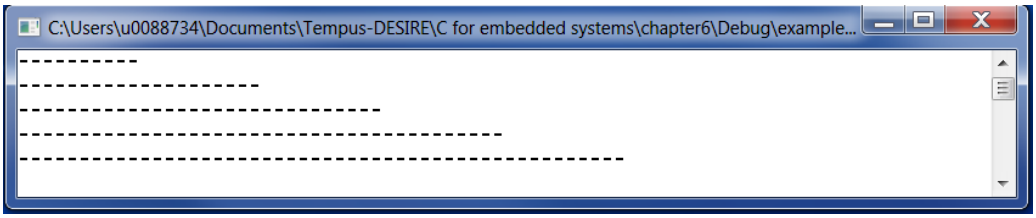
The function declaration only contains the types of the different parameters:

```
void <function_name>(type, type, …);
```

When the function is called, the parameters in the function definition are replaced by real arguments. Every argument is an expression that is evaluated during the function call. The resulting argument value is then used to initialize the formal parameters.

Example:
Write a program that prints several lines made out of '-' characters. The line lengths can vary.

```
1   /*
2      Use function to print out a line with different lengths
3   */
4
5   #include <stdio.h>
6
7   void line(int);            /* function declaration */
8
9   int main(void)
10  {
11     line(10);                /* function call */
12     line(20);
13     line(30);
14     line(40);
15     line(50);
16
17     return 0;
18  }
19
20  void line(int length)                /* function definition */
21  {
22     int i;
23
24     for (i = 0; i < length; i++)
25           printf("-");
26
27     printf("\n");
28  }
```




**Code 26: example void function with parameters**

The function declaration

```
void line(int);
```

tells the compiler that the function with name "`line`" has no return value (`void`) and expects an integer value as input (`int`).

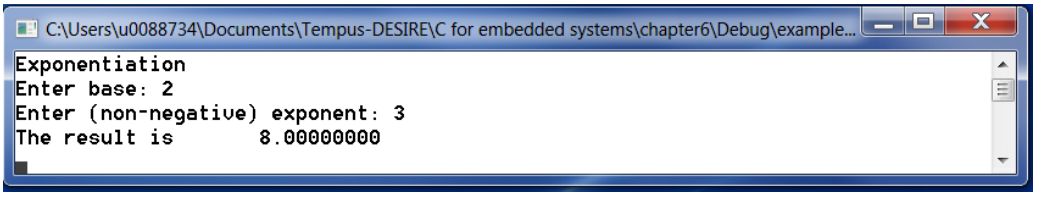
The function `line` is called with the statement "`line(40);`". The argument value 40 is copied to the argument `length` inside the function `line`. In this function, the `for` loop takes care of printing the correct amount of "-" characters.

### 6.2.3 Functions with return value.

As shown in Figure 28, a function can also produce a result that needs to be given back to the function caller. This function result is called the function-return-value and is written before the function name in both the function declaration and the function definition.

Example:
write a program that reads a real base and a non-negative integer exponent and calculates the exponentiation.

```
1   /*
2      example of function with return value
3   */
4
5   #include <stdio.h>
6   double exponentiation(double, int);     /* function declaration */
7
8   int main(void)
9   {
10     double a, m;
11     int n;
12     printf("Exponentiation \n");
13     printf("Enter base: ");
14     scanf("%lf%*c", &a);
15     printf("Enter (non-negative) exponent: ");
16     scanf("%d%*c", &n);
17     m = exponentiation(a, n);            /* function call */
18     printf("The result is %16.8f \n", m);
19     return 0;
20  }
21
22  /* function definition */
23  double exponentiation(double base, int exponent)
24  {
25     double result = 1.0;
26     int i;
27     for (i = 0; i<exponent; i++)
28     {
29            result *= base;
30     }
31     return result;
32  }
```




**Code 27: example function with return value**

The statement "`return result;`" inside the function `exponentiation`, returns the result to the function `main` where `exponentiation` was called. In this example, this result is assigned to the variable `m`.

**Remark**

When the return statement is executed, the return value is passed back to the caller function. On top, the program execution returns to the caller function!
As a result, statements written after the return statement in a function will be ignored!

## 6.3 Storage classes and scope of variables

Every variable has its own storage class and scope. The storage class indicates how long a variable remains valid while the scope gives information on where the variable can be accessed.

### 6.3.1 Storage class `auto` – local variables

Auto variables have automatic storage duration, meaning that they will be created upon entrance of the block in which they are defined, exist while the block is active and are destroyed when that block is exited.

The scope of these variables is local to the block in which they are defined. No block outside the defining block has direct access to automatic variables!

These variables may be declared with the keyword `auto` but this is not mandatory. If no storage class is written, the class `auto` is used as well. As such, if no keyword is added, the variables defined in a function have a scope local to this function!

### 6.3.2 Storage class `extern` – global variables

External variables have static storage duration, meaning that memory will be allocated when the program execution starts and will remain allocated until the program terminates.

An example of external variables are global variables. These variables are created by placing the variable declaration outside any function. In this case, the keyword `extern` does not need to be added. The variable will be external by default.

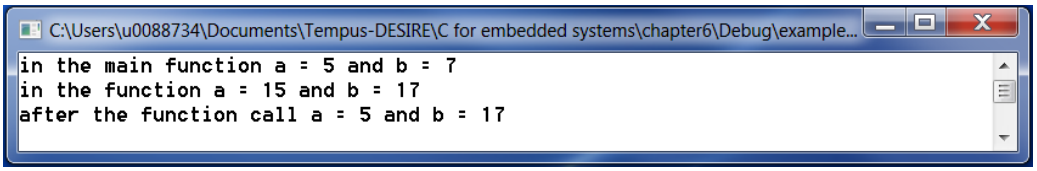
The scope of global variables is the entire source code following the declaration. If these variables are not initialized in the declaration, they will be initialized to zero automatically.

**Learning note**

Global values can be accessed and altered from everywhere in the program! Therefore avoid using global variables!!

The program in Code 28 demonstrates the difference between local and global variables. Note that, when entering the function, the local variable `a` belonging to the function has priority over the global variable `a`. Since the variable `b` is not declared inside the function, every calculation or assignment done with that variable, refers to the global variable `b`.

```
1   /*
2      Program with global and local variables
3      demonstration of the "scope" of a variable
4   */
5   int a, b;              /* global variables */
6   void f(void);          /* function declaration */
7
8   int main(void)
9   {
10     a = 5;
11     b = 7;
12     printf("in the main function a = %d and b = %d \n", a, b);
13     f();
14     printf("after the function call a = %d and b = %d \n", a, b);
15     return 0;
16  }
17
18  void f(void)
19  {
20     int a;     /* local variable, the global var "a" is invisible */
21     a = 15;
22     b = 17;
23     printf("in the function a = %d and b = %d \n", a, b);
24  }
```




**Code 28: difference between global and local variables**

**Learning note**

To maximize the reusability of functions, avoid all external dependencies in functions. This can be accomplished by using local variables only. Do **not** use global variables.

### 6.3.3 Storage class `register`

This is a special case of the storage class `auto`. If the word `register` is written in front of a variable declaration, it suggests the compiler to allocate an automatic variable in a high-speed CPU-register, if possible.

Of course, this only makes sense if speed is of outmost importance for that variable.

Example:

```
1   /*
2      Program with a register - variable
3   */
4   #define LIMIT 1000
5
6   int main(void)
7   {
8      register int i;            /* local variable of type register*/
9      for (i = 0; i<LIMIT; i++)
10     {
11            printf("%8d", i);
12     }
13     return 0;
14  }
```

**Code 29: register storage class**

### 6.3.4 Storage class `static`

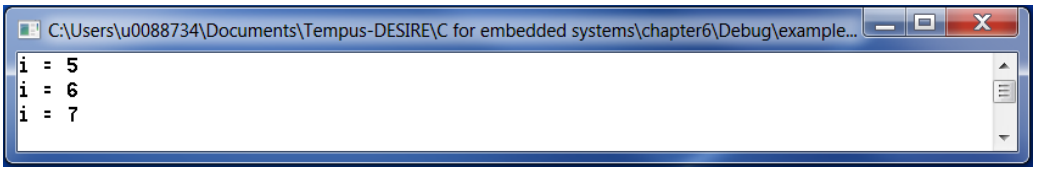
Like variables of the storage class `extern`, also `static` variables have a static storage duration, meaning that memory will be allocated when the program execution starts and will remain allocated until the program terminates.

The scope of these variables is often (but not necessary) limited! If a `static` variable is defined in a block, the scope of that variable is identical to the scope of automatic variables. Therefore, the variable will keep on existing after the execution of the code block, but none of the other code blocks can access that variable.

Static variables are initialized only once!

Code 30 shows an example of `static` variable usage.

```
1   /*
2      example with a static variable
3   */
4
5   void f(void);
6
7   int main(void)
8   {
9      f();
10     f();
11     f();
12     return 0;
13  }
14
15  void f(void)
16  {
17     static int i = 5;   /* static variable (local to function f) */
18     printf("i = %d \n", i);
19     i++;
20  }
```




**Code 30: storage class `static`**

## 6.4 Structured programming example

Write a program that reads the coefficients a, b and c of the quadratic equation $ax^2 + bx + c = 0$ and prints the roots.

To solve this problem, we will divide it in several smaller sub problems:

- if the coefficient a = 0, the equation is reduced to a linear equation. We will write a separate function to solve linear equations and one to solve quadratic equations. This is represented in the flowchart below:

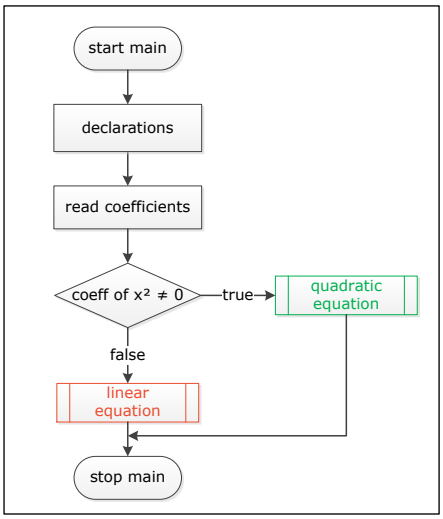



**Figure 30: structured programming example main function**

- The algorithm needed to solve a linear equation is shown in Figure 31:

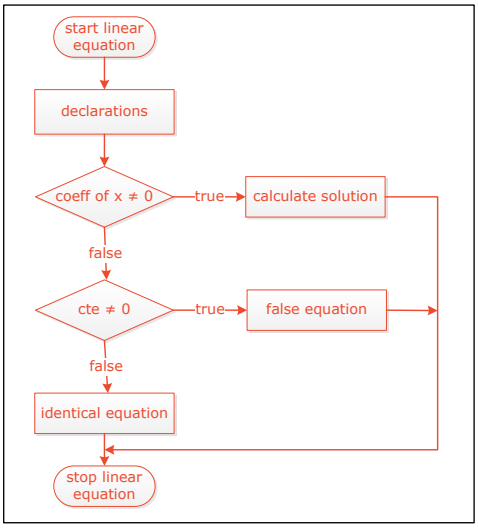



**Figure 31: structured programming example function linear equation**

- To solve a quadratic equation, the discriminant needs to be calculated. Depending on the value of the discriminant, we have 2 real, 1 real or 2 complex roots. Also these parts can be written in separate functions:

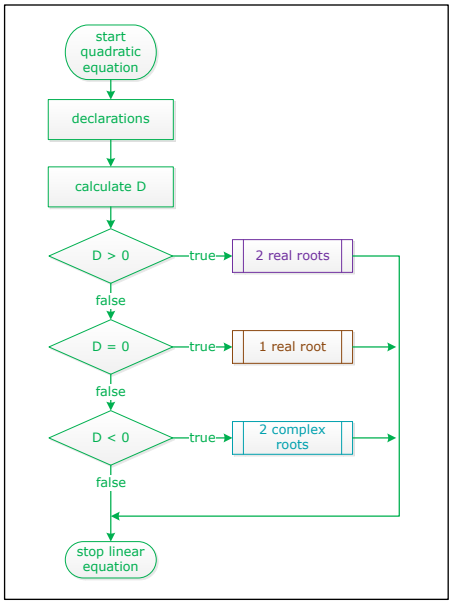



**Figure 32: structured programming example function quadratic equation**

- Finally, the 3 last functions can be visualized as follows:

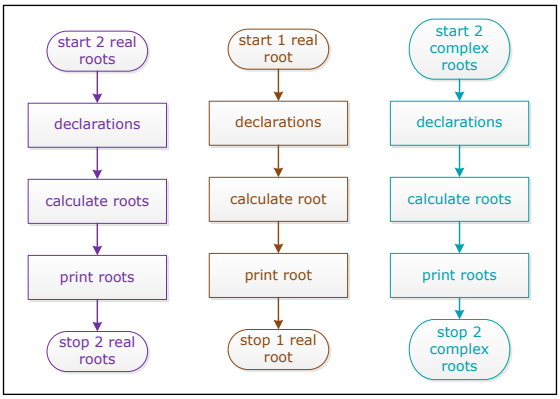



**Figure 33: structured programming example functions root calculation**

Translating into C code results in:

```
1   /*
2      solving a quadratic equation
3   */
4
5   #include<stdio.h>
6   #include<math.h>
7
8   /* function declarations*/
9   void linear(float, float);
10  void quadratic(float, float, float);
11  void tworealroots(float, float, float);
12  void oneroot(float, float);
13  void twocomplexroots(float, float, float);
14
```

```
15  int main(void)
16  {
17     float a, b, c;
18     printf("Solving a quadratic equation. \n");
19     printf("Enter the coefficients a, b and c ");
20     scanf("%f%f%f%*c", &a, &b, &c);
21
22     if (a)
23     {
24           quadratic(a, b, c);
25     }
26     else
27     {
28           linear(b, c);
29     }
30
31     return 0;
32  }
33
34  /* function definitions */
35  /* function to solve the linear equation ax+b=0 */
36  void linear(float a, float b)
37  {
38     if (a)
39     {
40           printf("linear equation with solution: %f \n", -b / a);
41     }
42     else
43     {
44           if (b) printf("False equation \n");
45           else printf("Identical equation \n");
46     }
47  }
48
49  /* function to solve a true quadratic equation*/
50  void quadratic(float a, float b, float c)
51  {
52     float d;
53     d = b * b - 4 * a * c;
54     if (d > 0) tworealroots(a, b, d);
55     if (d == 0) oneroot(a, b);
56     if (d < 0) twocomplexroots(a, b, d);
57  }
58
59  void tworealroots(float a, float b, float d)
60  {
61     float sqrt_d, x1, x2;
62     sqrt_d = sqrt((double)d);
63     x1 = (-b + sqrt_d) / 2 / a;
64     x2 = (-b - sqrt_d) / 2 / a;
65     printf("Two real roots: %f and %f \n", x1, x2);
66  }
67
68  void oneroot(float a,  float b)
69  {
70     printf("One root %f \n", -b / 2 / a);
71  }
72
73  void twocomplexroots(float a, float b, float d)
74  {
75     float re, im;
76     re = -b / 2 / a;
77     im = sqrt((double)(-d)) / 2 / a;
78     printf("2 complex roots: %f+%fi and %f-%fi \n",re,im,re,im);
79  }
```

**Code 31: structured programming example**

## 6.5 Exercises

**6.1.** Write a program that reads an angle in degrees and prints the corresponding sine. You can use standard functions.

**6.2.** Write a program that prints a table with 2 columns. The first column contains all angles from 0 till 360 degrees with steps of 30 degrees. The second column contains the corresponding sine values.

**6.3.** Write a program that calculates the square root of a number entered by the user.

**6.4.** Write a program that reads the lengths of the sides a and b of a right-angled triangle and prints the length of the hypotenuse c and one of the acute angles.

examples:

| | | | |
|---|---|---|---|
| input: 2 | 1 | output: 2.24 | 63° |
| input: 1 | 1.732 | output: 2.00 | 60° |

**6.5.** Write a program for a guessing game. First a random number between 1 and 100 is chosen by the program. Afterwards, the user can start guessing. If the guess was too high or too low, the program needs to print "too high" or "too low". This is repeated until the number was found. In the end, the program prints how many guesses the user needed to find the secret number.

**6.6.** Write a program that prints a table with 2 columns. The first column contains $x$ values from -5 till +5 with a step of 0.5. The second column contains the corresponding $y$ values according to the equation

$$y = 2x^2 + 2x - 3$$

Make sure the calculation of the y values is done in a separate function.

**6.7.** Write a function with header:

```
void printline( int number, char c)
```

that prints a line of `number` symbols `c` followed by a newline character. Write a main program that calls this function several times with different parameters.

example: the statement

```
printline(40, '*');
```

results in

```
****************************************
```

**6.8.** Write a function that reads an integer number in the interval [0 , 10] and returns that number as function return value. If the number is not in the correct interval, the function needs to ask a new number until a correct value was entered.
The main program is something like:

```
int main(void)
{
     int number;
     number = readnumber();
     printf("The number read is %d\n", number);
     return 0;
}
```

make sure you write the declaration of the function `readnumber()` before the `main` function and the definition of the function after the `main` function.

**6.9.** Write a function with header:

```
int readnumber(int lower_boundary, int upper_boundary)
```

that reads an integer number in the interval [lower_boundary, upper_boundary] and returns that number as function return value. Also in this case, the function can only stop asking an integer number if a correct value was entered.
Write a `main` function that calls this function and prints the resulting number.

**6.10.** Write a function with header:

```
double exponentiation(double base, int exponent)
```

that returns $base^{exponent}$ as function return value.
Write a `main` function that reads a base and exponent, that calls the function exponentiation and that prints the result. Do not use the standard function `pow`!

**6.11.** Write a function with header:

```
int gcd(int number1, int number2)
```

that returns the greatest common divisor of the numbers number1 and number2 as function return value.
Write a main program that reads 3 integer numbers, calculates the gcd of those 3 numbers and prints the result.
**Hint**: gcd(a, b, c) = gcd( a, gcd(b, c))

**6.12.** Write a program that reads a number of scores. The scores are all positive integer numbers. A negative number is entered to indicate that all scores were entered.
For each score, a bar with a length equal to the score is drawn. To this end, a predefined symbol needs to be printed as many times as the score. (for loop).
Write a function `draw_bar` that takes a score as input and draws a bar with corresponding length. The scores and the wanted symbol are read in the `main` function.

The screen dialog should look like:

```
What symbol would you like to use? =
Enter scores: 2 12 18 3 -5

score = 2         ==
score = 12        ============
score = 18        ==================
score = 3         ===
```

**6.13.** Write a program with the functions:
- `hello`: that welcomes the user and explains what is expected
- `main`: that asks the user to enter 5 times 2 numbers
- `sum`: that asks the user to enter the sum of the previously entered numbers and gives feedback to the user
- `goodbye`: that thanks the user for his/her cooperation

```
Welcome, this program will ask you to solve 5 sums.

enter 2 numbers < 100: 15 16
what is the sum of 15 and 16? 31
according to you, the sum of 15 and 16 equals 31. That
is correct

enter 2 numbers < 100: 26 32
what is the sum of 26 and 32? 50
according to you, the sum of 26 and 32 equals 50. That
is not correct
…
Thanks for your cooperation.
```

**6.14.** Since a year is not exactly 365 days, we have a leap year once every 4 years except if the year is dividable by 100. If the year is dividable by 400, the year is considered as a leap year anyway.
Write a function with header:

```
int isLeapYear( int year)
```

that determines if a year is a leap year or not and gives a different function return value for both cases.

Write also a function with header:

```
int numberOfDays(int month, int year)
```

that calculates the number of days in the month `month` of the year given.

Write a main function that reads a month and a year and prints the number of days in that month of that year.
examples:
month 2 of 2000 has 29 days
month 2 of 1900 has 28 days
month 1 of 1950 has 31 days

**6.15.** Write a function with as parameters 3 integer numbers that represent a day, month and year. This function calculates and returns a factor according to following formula:

$$factor = 365 * year + day + 31 * (month - 1) + \left[\frac{year - 1}{4}\right] - \left[\left[\frac{year - 1}{100} + 1\right] * \frac{3}{4}\right]$$

for the months January and February
with [x] being the integer part of x

$$factor = 365 * year + day + 31 * (month - 1) - [0{,}4 * month + 2{,}3] + \left[\frac{year}{4}\right] - \left[\left[\frac{year}{100} + 1\right] * \frac{3}{4}\right]$$

for the months March till December.

Write a `main` function that reads 2 dates, calculates the number of days in between these 2 dates by calculating the difference between the 2 factors of the corresponding dates.

**6.16.** Consider the first quadrant of a circle in a square with side 1. If you generate a large amount of (x, y) coordinates with x and y belonging to the interval [0, 1], you have a collection of points belonging to the square. If you now count all points that belong to the quadrant of the circle with equation $x^2 + y^2 < 1$, and divide this amount by the total amount of points generated, you will find approximately the number $\frac{\pi}{4}$.

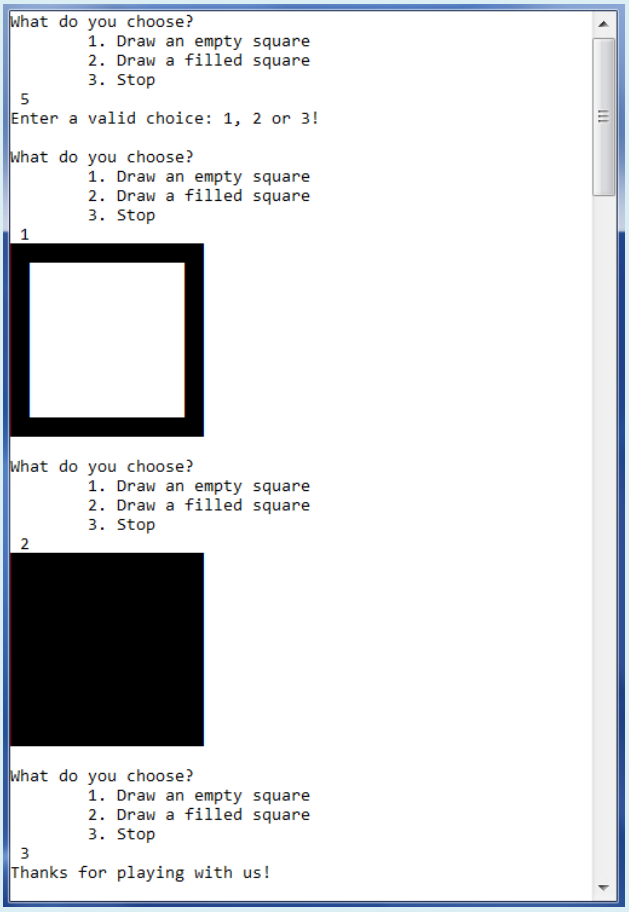
Write a program that generates 100 000 points and uses it to calculate the number $\pi$ approximately. Run the program several times and compare the results. What if you increase the number of points?

**6.17.** Write a program that asks the user to choose between 3 options:

1. Draw an empty square
2. Draw a filled square
3. Stop

The program repeats itself until the user chooses the option 3 (Stop). Use the functions '`empty_square`' and '`filled_square`'

The screen dialog should look like:



# 7 Arrays

**Objectives**

This chapter explains the use of the array data structure. You will learn how to declare and initialize an array and how to refer to one array element. Also passing arrays to functions as arguments is discussed.

## 7.1 Definition

An array is a data structure, used to store a collection of elements of the same type. Although an array is used to store a collection of data, it is often more useful to think of an array as a collection of variables of the same type. Each of those elements are identified by the same array name but with a different array index (see Figure 34).

## 7.2 Array declaration

In C all variables must be declared before they are used. This is needed to make sure the correct amount of memory is reserved. Arrays also occupy memory and as such they need to be declared as well. To allow for correct computation of the total amount of memory needed, the data type of each element as well as the number of elements the array will contain, is specified. In its most general form, the declaration of an array can be written as:

```
type <array_name>[number_of_elements];
```

```
int a[10];          // declares an array of 10 integer elements
```

All arrays consist of contiguous memory locations. The lowest memory address corresponds to the first element of the array and is referred to with index `0`, the highest address corresponds to the last element and can be accessed with index "`number_of_elements - 1`". The array "`a`" with 10 integers can be visualized as shown in Figure 34:

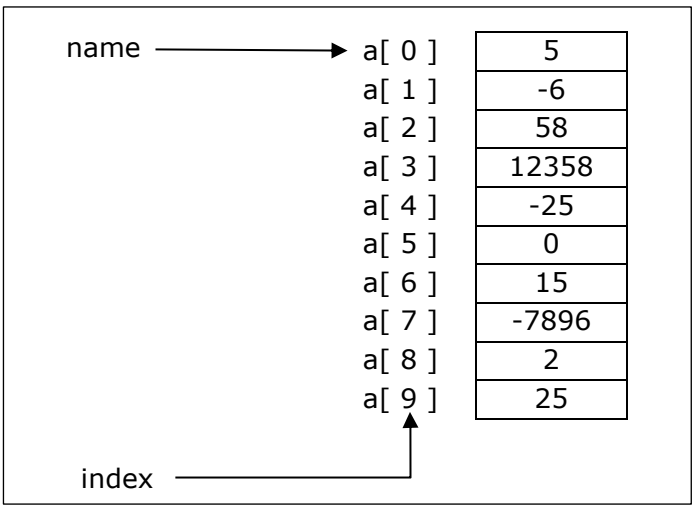



**Figure 34: array 'a' with 10 integers**

As array index also variables or expressions can be used. To access a valid array element, that variable or expression needs to evaluate to a positive integer belonging to the interval [0, number_of_elements – 1].

## 7.3 Array initialization

Entering a value into a specific array element can be done by the statement:

```
name[index] = value;
```

Hence, initialization of the array can be done as shown in the code below:

```
int a[5];
a[0] = 25;
a[1] = -2;
a[2] = 125;
a[3] = -25;
a[4] = 7;
```

Arrays can also be initialized in the definition of the array with an initializer list as follows:

```
int a[5] = {25, -2, 125, -25, 7};
```

Or, if all values are to be set to the same number (zero for instance) a for loop can be used:

```
int i;
int a[10];

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

## 7.4 Array usage

If we want to actually use arrays, we need the possibility to both write to and read from the array elements. Reading a value from a specific array element can be done by the statement:

```
variable = name[index];
```

In most programs, the elements of the array will not only be of the same type but also of the same kind. Meaning that they will have similar meaning. As a result, the elements of the array will often be treated in the same way. This can be achieved using a for loop as can be seen in following example.

Example:
write a program that reads 100 integers and prints them in reverse order.

```
1   /*
2      read 100 integers and print them in reverse order
3   */
4   #include <stdio.h>
5   #define SIZE 100
6
7   int main(void)
8   {
9      int numbers[SIZE];
10     int i;
11
12     printf("Enter %d integers: \n", SIZE);
13     //read the integers one by one in a for loop
14     for (i = 0; i < SIZE; i++)
15     {
16            printf("Enter integer %3d: ", i + 1);
17            scanf("%d%*c", &numbers[i]);
18     }
19
20     printf("In reverse order: \n");
21     //print the array elements starting from the last one down
22     for (i = SIZE - 1; i >= 0; i--)
23     {
24            printf("%8d", numbers[i]);
25     }
26
27     printf(" \n");
28     return 0;
29  }
```

**Code 32: array usage example**

Note the difference between the `scanf` and the `printf` statements!
As indicated before, you can consider an array as a group of variables that belong together. Taking this approach into account, one array element (`numbers[i]`) can be treated like a normal variable. Therefore, if the address of the variable is needed, like in the `scanf` function, the address operator `&` needs to be added, whereas no extra operator is needed in the `printf` function. The `printf` function only needs the value of the variable, not the address.

Before the main function definition, an extra precompiler directive is added:

```
#define SIZE 100
```

This allows to specify the size of the array with a symbolic constant. It makes programs more scalable. If for instance in the above example the number of integers needs to be changed into 5 instead of 100, it is sufficient to change only the `#define` preprocessor directive.

**Learning note**

Use symbolic constants for all array sizes in your program

**Common mistake**

Do not end the #define preprocessor directive with a semicolon! This is not a C statement!

## 7.5 Operations on arrays

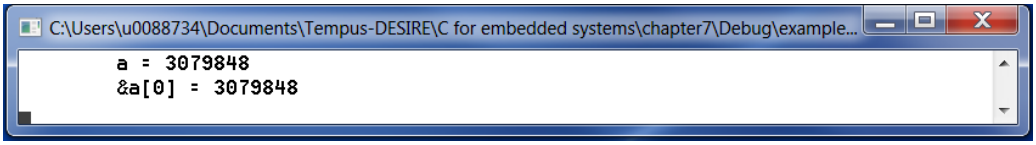
Consider 2 arrays a and b, both consisting of 5 elements of the type int. If we want to copy the elements of array a into array b, it cannot simply be done by assigning one array to the other!!

```
b = a;
```

does not work if a and b are arrays!

The name of an array points to a series of variables rather than one single variable. It can be seen as the starting address of the array as shown in Code 33 and Figure 35. Since the starting addresses of the arrays are fixed during program execution, the address of array b cannot be changed inside the program!

```
1   #include <stdio.h>
2   #define SIZE 5
3
4   int main(void)
5   {
6      int a[SIZE];
7
8      printf("\ta = %d\n\t&a[0] = %d\n", a, &a[0]);
9
10     return 0;
11  }
```




**Code 33: array name**

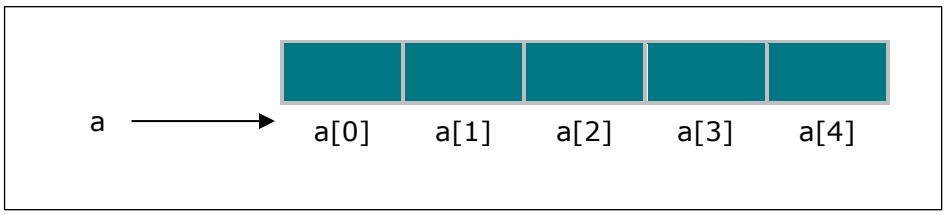



**Figure 35: array name**

As a result, operations on arrays like copy, addition, comparison, … always need to be done looping over all array elements.

## 7.6 Passing arrays to functions

To pass an array to a function, simply use the array name without brackets as function argument:

```
int a[5];
function(a);
```

The name of an array evaluates to the starting address of that array. Therefore, instead of passing the full array to the function, actually, the array starting address is passed to the function. As a result, the formal parameter in the function gets the same starting address as the one in the calling function. Or, in other words: the array in the function is the same as the one in the calling function. So, when the called function modifies array elements in its function body, it is modifying the elements of the original array! This is illustrated in Figure 36.

Since all information needed in the called function is the starting address of the original array, there is no need to specify the length of the array in the function declaration and definition. Therefore, often the array length is passed on to the function as an extra parameter (see Figure 36).

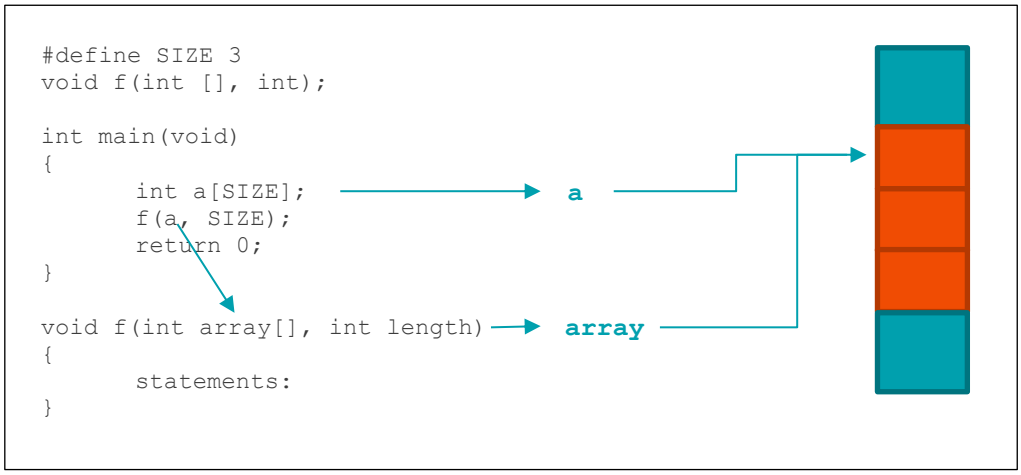



**Figure 36: passing arrays to functions**

Example:
Write a program that reads 100 integers and prints them in reverse order, using different functions to read and print the integers.

```
1   /*
2      read 100 integers and print them in reverse order
3      use different functions to read and print the integers
4   */
5   #include <stdio.h>
6   #define SIZE 5
7
8   /* function declarations*/
9   void readnumbers(int [], int);
10  void printnumbers(int [], int);
11
12  int main(void)
13  {
14     int numbers[SIZE];
15     readnumbers(numbers, SIZE);      //pass array name to function
16     printnumbers(numbers, SIZE);     //pass array name to function
17     return 0;
18  }
```

```
19  /*function definitions*/
20  void readnumbers(int x[], int length)
21                       //x and numbers refer to same memory location
22  {
23     int i;
24     printf("Enter %d integers: \n", length);
25
26     for (i = 0; i < length; i++)
27     {
28            printf("Enter number %3d: ", i + 1);
29            scanf("%d%*c", &x[i]);
30     }
31  }
32
33  void printnumbers(int r[], int length)
34                       //r and numbers refer to same memory location
35  {
36     int i;
37     printf("In reverse order: \n");
38
39     for (i = length - 1; i >= 0; i--)
40     {
41            printf("%8d", r[i]);
42     }
43
44     printf(" \n");
45  }
```

**Code 34: array with functions example**

**Common mistake**

Since all array modifications done by the called function directly modify the elements of the original array, there is no need to return an array explicitly from a function!!

**Remark**

If only 1 array element needs to be passed on to a function, you do not need to pass the full array. 1 element can be treated like a normal variable. Passing the value of that 1 element can be done by using array[i] as function argument.

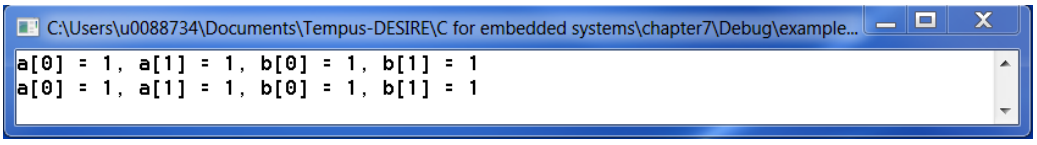
## 7.7 Array boundaries

In the array declaration, you need to specify the size of the array. For instance "`int x[5];`" declares an array with 5 integers that can be addressed as `x[0]`, `x[1]`, `x[2]`, `x[3]` and `x[4]`.

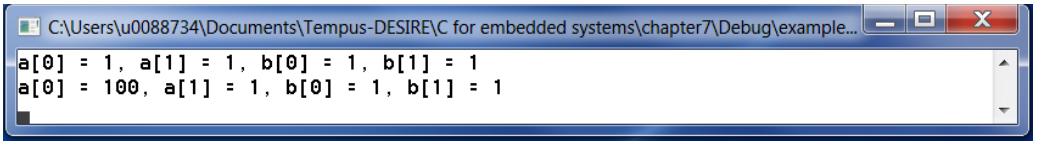
Unfortunately, the C compiler only verifies if every index used is an integer. It will not flag an error if an index outside the array boundaries is used! As a result, memory locations outside the memory reserved for the array will be used, which leads to dangerous and unpredictable behavior. This is illustrated in Code 35.

```
1   int main(void)
2   {
3      int b[2];
4      int a[2];
5      b[0] = 1;
6      b[1] = 1;
7      a[0] = 1;
8      a[1] = 1;
9      printf("a[0]=%d,a[1]=%d,b[0]=%d,b[1]=%d\n",a[0],a[1],b[0],b[1]);
10
11     /*unpredictable behavior: b[2] is an invalid array location!*/
12     b[2] = 100;
13     printf("a[0]=%d,a[1]=%d,b[0]=%d,b[1]=%d\n",a[0],a[1],b[0],b[1]);
14
15     return 0;
16  }
```

With following expected result:




Unfortunately, running the same program on another machine or at a different time can just as well result in:




**Code 35: array usage outside of array boundaries**

**Common mistake**

Not respecting the boundaries of an array is a common mistake that leads to errors that are difficult to find!

**Learning note**

Only use indices in the interval [0, number_of_elements – 1]

## 7.8 Programming examples using arrays

### 7.8.1 The sieve of Eratosthenes

The sieve of Eratosthenes is an efficient method to identify prime numbers. We will use it to find all prime numbers < 1000. The algorithm can be described as follows:

1. Make an array with 1000 elements and fill every element with the number 1 (except for the elements with index 0 and 1, since 0 and 1 are for sure no prime numbers).
2. Start with the element with index 2 (2 is a prime number, so this element remains 1).
3. Mark all multiples of two. To this end, change the content of every array element with an index that is a multiple of 2, from 1 into 0.
4. Find the first array element with an index > 2 that was not marked yet.
5. Repeat the algorithm from step 2.

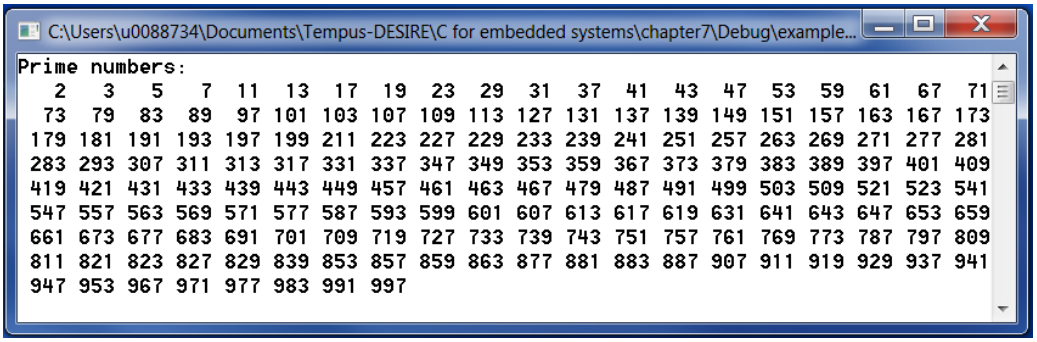
To make the program more readable and better structured, we will divide it in smaller sub problems or functions. In the solution of Code 36 the following functions are used:

- `initialize`: to fill all sieve elements with 1
- `mark_multiples`: to mark out all elements with a non-prime index
- `print_sieve`: to print all prime numbers

Translating into C code results in:

```
1   /*
2      find all prime numbers < MAX using the sieve of Eratosthenes
3   */
4   #include <stdio.h>
5   #define MAX 1000
6
7   void initialize(int[]);
8   void mark_multiples(int[]);
9   void print_sieve(int[]);
10
11  int main(void)
12  {
13     int sieve[MAX];
14
15     initialize(sieve);
16     mark_multiples(sieve);
17     print_sieve(sieve);
18     return 0;
19  }
20
21  void initialize(int sieve[])
22  {
23     int i;
24
25     sieve[0] = sieve[1] = 0;
26
27     for (i = 2; i<MAX; i++)
28            sieve[i] = 1;
29  }
```

```
30  void mark_multiples(int sieve[])
31  {
32     int i, j;
33
34     for (i = 2; i<MAX; i++)
35     {
36            if (sieve[i])
37            {
38                   for (j = 2 * i; j < MAX; j += i)
39                          sieve[j] = 0;
40            }
41     }
42  }
43
44  void print_sieve(int sieve[])
45  {
46     int i;
47
48     printf("Prime numbers:\n");
49
50     for (i = 2; i < MAX; i++)
51     {
52            if (sieve[i])
53                   printf("%4d", i);
54     }
55
56     printf("\n");
57  }
```



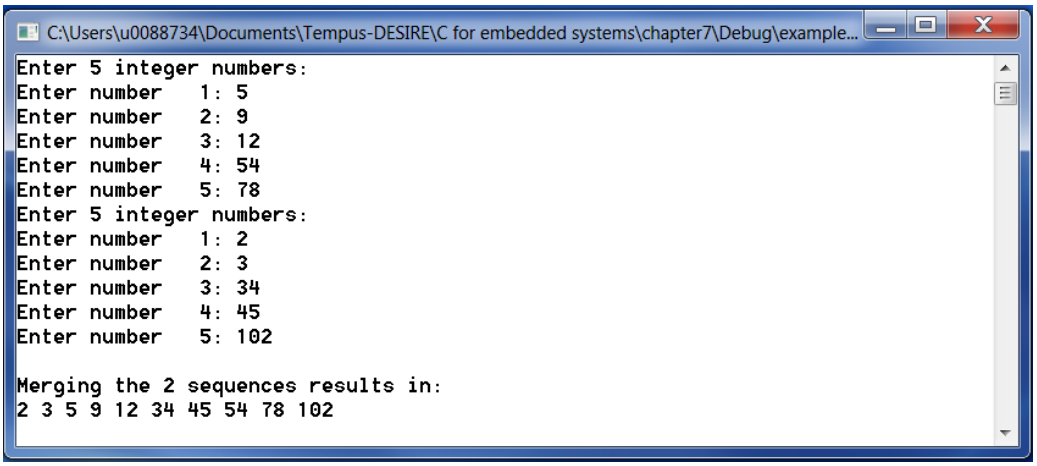

**Code 36: sieve of Eratosthenes**

## 7.8.2 Merging arrays

As a second example, we will write a program that reads 2 lists of 5 ascending integer numbers. Afterwards, the 10 numbers are printed together in ascending order. To this end, we will use twice the same function `ReadArray`. Printing is done with another function (`PrintArray`).

```
1   /*
2      merging arrays
3   */
4   #include <stdio.h>
5   #define SIZE 5
6
7   void ReadArray(int[], int);
8   void PrintArray(int[], int[], int);
9
```

```
10  int main(void)
11  {
12
13     int a[SIZE];
14     int b[SIZE];
15
16     ReadArray(a, SIZE);
17     ReadArray(b, SIZE);
18     PrintArray(a, b, SIZE);
19
20     return 0;
21  }
22
23  void ReadArray(int x[], int length)
24  {
25     int i;
26
27     printf("Enter %d integer numbers:\n", length);
28
29     for (i = 0; i < length; i++)
30     {
31            printf("Enter number %3d: ", i + 1);
32            scanf("%d%*c", &x[i]);
33     }
34  }
35
36  void PrintArray(int a[], int b[], int length)
37  {
38     int i, j;
39
40     i = 0;
41     j = 0;
42
43     printf("\nMerging the 2 lists results in:\n");
44
45     while (i < length && j < length)
46     {
47            if (a[i] < b[j])
48            {
49                   printf("%d ", a[i]);
50                   ++i;
51            }
52            else
53            {
54                   printf("%d ", b[j]);
55                   ++j;
56            }
57     }
58
59     while (i < length)
60     {
61            printf("%d ", a[i]);
62            ++i;
63     }
64
65     while (j < length)
66     {
67            printf("%d ", b[j]);
68            ++j;
69     }
70
71     printf("\n");
72  }
```




**Code 37: merging arrays**

## 7.9 Exercises

**7.1.** Write a program that reads the temperatures of a whole week into 1 array '`temperature[]`' and prints the mean temperature for that week.

```
Enter temperature for day 0: 5
Enter temperature for day 1: 5
Enter temperature for day 2: 5
Enter temperature for day 3: 5
Enter temperature for day 4: 6
Enter temperature for day 5: 6
Enter temperature for day 6: 6

The mean temperature for this week is 5.43
```

**7.2.** Change the previous exercise such that all days with a temperature warmer than 10°C are printed.

```
Enter temperature for day 0: 1
Enter temperature for day 1: 8
Enter temperature for day 2: 10
Enter temperature for day 3: 12
Enter temperature for day 4: 15
Enter temperature for day 5: 14
Enter temperature for day 6: 9

All days with a temperature > 10°C:
day 3
day 4
day 5
```

**7.3.** Repeat exercise 7.1 using 2 functions. Make a function to read the temperatures and a separate function to calculate the mean temperature. Printing the mean must be done in the main program.

**7.4.** Repeat exercise 7.2 but this time, use 3 separate functions to read the temperatures ("`ReadTemp`"), to determine which days are warmer than 10°C ("`Calculate`") and to print the days found ("`PrintDays`"). Extra: ask the user to enter a temperature limit.

**7.5.** Write a program with following main function:

```
#include <stdio.h>
#define SIZE 12
#define COLUMNS 3

void ReadArray(int [], int);
void PrintMatrix(int [], int, int);

int main(void)
{
    int a[SIZE];
    ReadArray(a, SIZE);
    PrintMatrix(a, SIZE, COLUMNS);
    return 0;
}
```

- The function `ReadArray` reads 12 (`SIZE`) numbers and stores them in the one dimensional array `a`
- The function `PrintMatrix` prints the 12 numbers on 4 lines of 3 (`COLUMNS`) numbers each

By adapting only `SIZE` and `COLUMNS` the program must also be able to print for instance 20 numbers in a 5 x 4 matrix.

**7.6.** Change exercise 7.5 such that after every row, the sum of all elements in that row is printed.

**7.7.** Change exercise 7.6 such that under every column, the sum of all elements in that column is printed.

**7.8.** Write a program with a `main` function and 3 extra functions.

- In the `main` function an array of 100 integers is declared and the 3 extra functions are called.
- The first function reads 1 integer in the interval [0, 100]. This integer represents the effective number of elements the array will contain. This integer needs to be returned to the main program.
- The second function reads the correct number of integers and stores them in the array.
- The third function prints the previously read integers.

The main function looks like:

```
int main(void)
{
    int row[MAX];
    int size;
    size = ReadSize(0, MAX);
    ReadArray(row, size);
    PrintArray(row, size);
    return 0;
}
```

**7.9.** Same as exercise 7.8 but without asking the user to enter the effective number of integers. The user enters the integers and ends with 999 to indicate the end of the reading process. 999 can NOT be stored in the array!

The main function looks like:

```
int main(void)
{
    int row[MAX];
    int size;
    size = ReadArray(row);
    PrintArray(row, size);
    return 0;
}
```

**7.10.** Add an extra function to exercise 7.9 that calculates the mean value of all entered integers and returns that mean value to the `main` function. The mean must be printed in the `main` function.
The header of this extra function could be like:

```
double CalcMean(int row[], int size)
```

**7.11.** Repeat exercise 7.10 but this time, write a function that searches the max value of all integers entered and returns that max value to the `main` function.

**7.12.** Write a program with functions that merges 2 ordered rows (ordered from small to large) into 1 ordered row.

```
Enter ordered row: 5 9 12 54 78
Enter ordered row: 2 3 34 45 102

The merged row is: 2 3 5 9 12 34 45 54 78 102
```

**7.13.** Add an extra function to exercise 7.9 or 7.10 such that before printing, the largest and smallest integers are interchanged. Changing places needs to happen inside the array, do not use a second array. Using an auxiliary variable is allowed.

**7.14.** Write a program with functions:
- a first function reads 20 numbers and stores them in an array.
- a second function makes sure none of the numbers in the array occurs more than once. To this end, the second, third, … occurrence of a number is removed from the array and all other array elements are shifted to the left.

```
Enter a list of 20 numbers:
2 5 3 4 6 5 3 4 6 7 2 4 4 5 3 2 1 6 1 7

Following numbers are stored in the cleaned up array:
2 5 3 4 6 7 1
```

**7.15.** Write a program that generates 6 different random numbers in the interval [1, 42]. Use an array to store the numbers and to make sure all 6 numbers are different.

**7.16.** Write a program with functions. A first function reads 2 rows of maximal 10 positive integers. The reading process stops when a 0 or 10 integers are entered. (do not store the number 0!)
A second function checks if both rows contain numbers that are equal.

```
Enter a row of max 10 integers (stop with 0):
4 4 12 9 5 0
Enter a row of max 10 integers (stop with 0):
9 9 4 12 4 5 9 12 4 4

The rows:
4 4 12 9 5 0
and
9 9 4 12 4 5 9 12 4 4
contain equal numbers.
```

**7.17.** Write a program that simulates the rolling of 2 dice. Rolling 2 dice always results in a value between 2 and 12. How many times will every possible value occur if the dice are rolled 400 times?
- declare an array in the main function to keep the occurrences for every possible value
- a first function simulates 400 rolls with 2 dice and calculates the values. (hint: rand()%6+1 results in a random number from 1 to 6, and as such simulates the rolling of 1 die)
- A second function prints the occurrences.
- extra: print a horizontal bar chart with the occurrences. Use the symbol "#" to draw the bars. Use a function DrawLine to accomplish this part of the program

# 8 Strings

**Objectives**

In this chapter, the concept of arrays is used to define strings. You will learn how to declare, initialize and manipulate strings. Passing strings to functions will be explained as well as using special string functions.

There is no special data type to define strings in C. Instead, a string is represented as a sequence of characters, ended by a special string-termination character called the null byte (`'\0'`).

## 8.1 String constant

A string constant consist of a series of characters enclosed in double quotes. It may consist of any combination of digits, letters, escaped sequences and spaces.

In memory, it is stored as a sequence of characters followed by a null byte as shown in the example of Figure 37. In the compiled version of a program, the string constant will be replaced by the starting address of that string constant in memory. Therefore, a string must be seen as an address.

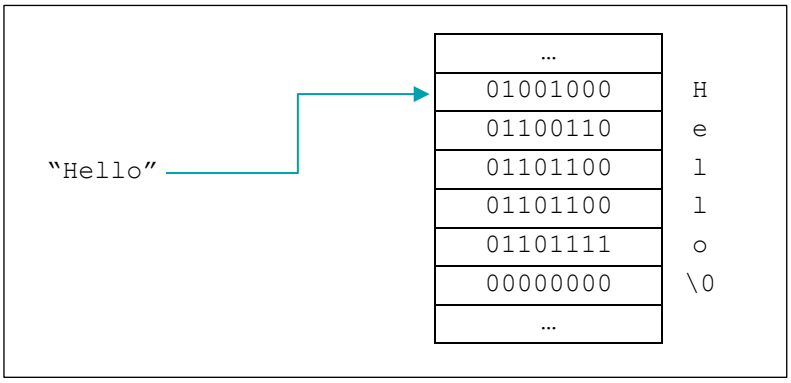



**Figure 37: string constant**

## 8.2 String variable

A string variable is basically an array of characters that must be large enough to hold all symbols of the strings you want to store and 1 extra symbol for the null byte.

The name of a string variable is the starting address of the memory reserved for the character array. The string declaration nails down to a declaration of a character array as shown in Figure 38.

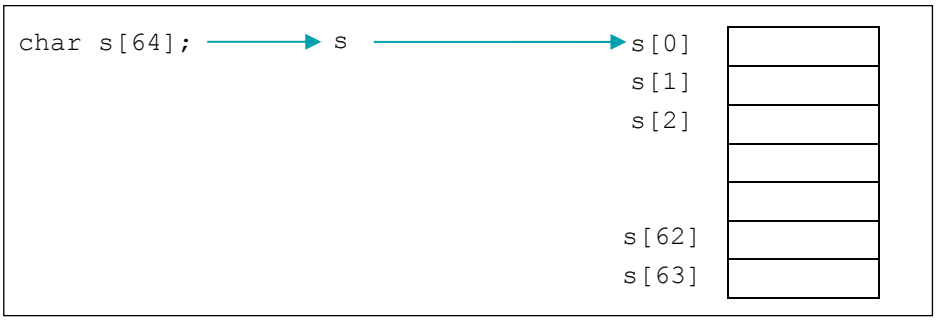



**Figure 38: string variable**

Assigning value to a string variable can be done in different ways:

1. assign individual characters to the string variable:

```
s[0]= 'a';
s[1]= 'b';
s[2]= 'c';
s[3]= '\0';        or s[3] = (char)0;
```

   Remark that the individual characters get single quotes! With double quotes, `"a"` represents a string consisting of the character a and the null byte.

2. Using string functions:

```
strcpy(s, "abc");
```

   The function `strcpy`, assigns the characters a, b and c to the string `s` and adds a null byte after the last character.

3. Reading a string from the keyboard:

```
scanf("%s%*c", s);
gets(s);
```

   both `scanf` and `gets` will read the user input and store it in the string `s`, adding a null byte at the end. `scanf` will stop reading at a space, tab or newline while `gets` will stop only at a newline character.
   **Remark** that in the scanf function, there is no address operator (`&`) before the name of the string `s`. This is because a string is an array of characters and the name of an array represents the starting address of that array. As such the name `s` already evaluates to the string's starting address so no `&` is needed here.

4. Initialization together with declaration:

```
char s[20] = "abc";
```

   while compiling, memory will be reserved for an array of 20 bytes. The first 3 elements of that array will be filled with the characters 'a', 'b' and 'c'. The forth element will be filled with a null byte.

**Common mistake**

None of the methods mentioned above will verify the string boundaries. It is your own responsibility to make sure the string variable is large enough to hold the strings you want to store + a null byte!

**Common mistake**

Assigning strings with a simple assignment operator is not possible!!
The statement

```
s = "abc";
```

cannot be used since both the string variable `s` and the constant string `"abc"` represent fixed addresses that cannot be changed!

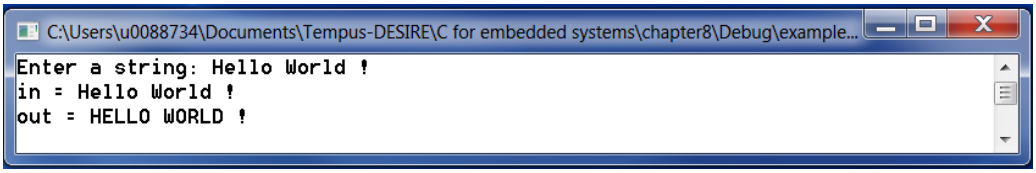
## 8.3 Passing strings to functions

Since strings are arrays of characters, strings are passed to functions like any other array:

- The formal function parameter gets the starting address of the string during the function call.
- When the called function modifies the string inside its function body, the original string in the calling function will also be changed.

Example:
Write a program that reads a string from the keyboard and converts it to capital letters. Use a separate function that converts the individual characters into capitals.

```
1   /*
2      convert lowercase letters into uppercase letters
3   */
4   #include <stdio.h>
5   void convert(char[], char[]);
6
7   int main(void)
8   {
9      char in[32];
10     char out[32];
11     printf("Enter a string: ");
12     gets(in);
13     convert(in, out);
14     printf("in = %s \nout = %s \n", in, out);
15     return 0;
16  }
17
18  void convert(char in[], char out[])
19  {
20     int i = 0;
21     while ((out[i] = in[i]) != '\0')
22     {
23            if (out[i] >= 'a' && out[i] <= 'z') out[i] += 'A' -'a';
24                   i++;
25     }
26  }
```




**Code 38: passing strings to functions**

Note that the conditional expression in the while loop consists of 2 parts:

```
(out[i] = in[i]) != '\0'
```

First the character stored in `in[i]` is copied to `out[i]`. Afterwards this symbol is compared to the null byte to detect the end of the string.

Converting from small letters to capital letters is done based upon the fixed difference in ASCII values between lowercase and uppercase letters. An ASCII table can be found in Attachment 2.

## 8.4 String functions

In this section a possible implementation of some commonly used string functions is shown. Of course, these functions can be included via "string.h", so you do not need to copy these codes into your programs.

### 8.4.1 `strlen`

`strlen(s);` returns the length of string `s` (null byte not included)

```
1   int strlen(char s[])
2   {
3       int i=0;
4
5       while( s[i] != '\0')
6           ++i;
7
8       return i;
9   }
```

**Code 39: string function `strlen`**

### 8.4.2 `strcpy`

`strcpy(s,t);` copies string `t` into string `s` (null byte is also copied)

```
1   int strcpy(char s[], char t[])
2   {
3       int i=0;
4
5       while((s[i] = t[i]) != '\0')
6           ++i;
7   }
```

**Code 40: string function `strcpy`**

### 8.4.3 strcmp

`strcmp(s,t);` returns

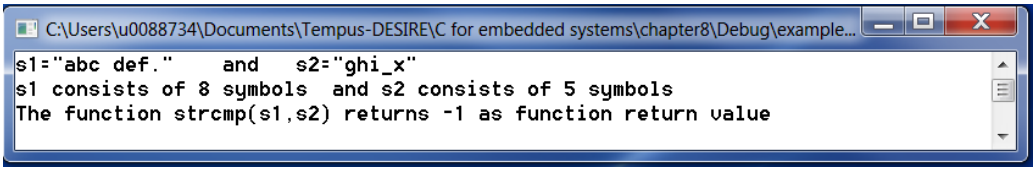
- an integer < 0 if string `s` < string `t` (alphabetically)
- 0 if string `s` and string `t` are equal
- an integer > 0 if string `s` > string `t` (alphabetically)

```
1   int strcmp(char s[], char t[])
2   {
3       int i;
4
5       for(i = 0; s[i] == t[i]; i++)
6       {
7           if( s[i] == '\0')
8               return 0;
9       }
10      return s[i] - t[i];
11  }
```

## 8.5 Programming examples using strings

### 8.5.1 Demonstration of several string functions

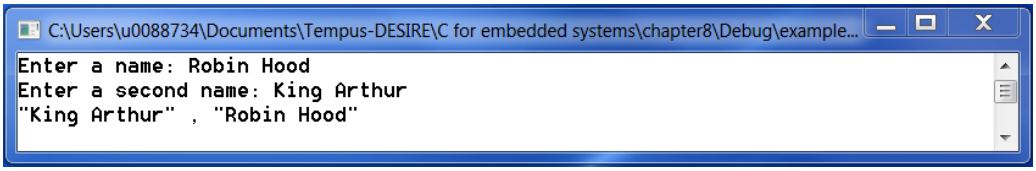
```
1   /*
2      using string functions from the standard library "string"
3   */
4   #include <stdio.h>
5   #include <string.h>
6
7   int main(void)
8   {
9      char s1[32];
10     char s2[32];
11
12     strcpy(s1, "abc def.");
13     strcpy(s2, "ghi_x");
14
15     printf("s1=\"%s\"    and   s2=\"%s\"\n", s1, s2);
16     printf("s1 consists of %d symbols  and s2 consists of %d
               symbols\n", strlen(s1), strlen(s2));
17     printf("The function strcmp(s1,s2) returns %d as function return
                value\n", strcmp(s1, s2));
18
19     return 0;
20  }
```




**Code 41: string example 1**

### 8.5.2 Sorting 2 strings alphabetically

```
1   /*
2      Printing 2 strings alphabetically
3   */
4   #include <stdio.h>
5   #include <string.h>
6
7   int main(void)
8   {
9      char name1[32];
10     char name2[32];
11     int comp;
12
13
14     printf("Enter a name: ");
15     gets(name1);
16     printf("Enter a second name: ");
17     gets(name2);
18
19     comp = strcmp(name1, name2);
20
21     if (comp == 0)
22     {
23            printf("\"%s\" and \"%s\" are equal\n", name1, name2);
24     }
25     else
26     {
27            if (comp < 0)
28            {
29                   printf("\"%s\" , \"%s\"\n", name1, name2);
30            }
31            else
32            {
33                   printf("\"%s\" , \"%s\"\n", name2, name1);
34            }
35     }
36
37     return 0;
38  }
```




**Code 42: printing 2 strings alphabetically**

**Remark**

The function `strcmp` determines the alphabetical order of strings based upon the ASCII values of the individual characters. Therefore, alphabetic ordering with strings containing both small letters and capitals or strings containing special characters or spaces will not always give a correct result. To avoid this, first convert the names into strings containing only capitals and remove all spaces and special characters before sorting.

## 8.6 Exercises

**8.1.** Write a program with functions.
- The first function reads a name.
- The second function prints the name.

**8.2.** Read 3 words separately into 3 different strings. Make a 4th string that contains the 3 words separated by a space and print the content of the 4th string.

```
enter the first word: this
enter the second word: is
enter the third word: it

this is it
```

**8.3.** Write a function with a string as parameter and an integer 1 (true) or 0 (false) as return value. The function tests whether the entered string is a palindrome or not. A palindrome is a word that reads the same backward and forward. (ex: noon, radar, rotor, racecar, …)
Write also a `main` function to test it.

**8.4.** Repeat exercise 8.3 but this time for palindrome sentences.
examples: "Was it a car or a cat I saw?" or "Eva, can I stab bats in a cave?" (note that punctuation, capitalization and spaces are ignored)

**8.5.** Repeat exercise 8.3 but this time for sentences that form a word based palindrome.
examples: "He says: it is true, true is it, says he"

**8.6.** Write a program that reads a string and prints every word of that string separately on a new line.

**8.7.** Write a program that reads 2 names and prints them alphabetically.

**8.8.** Repeat exercise 8.7 but now with 3 names.

**8.9.** Repeat exercise 8.8 but make sure also following names are sorted correctly: O'Neil, Mac Alastair, Macbride, mac Caba, O Neal, Orman
Hint: use your own string compare function that first copies the strings in capitals and without punctuation and spaces and uses `strcmp` on the copies)

# 9 Multidimensional arrays

**Objectives**

In this chapter the concept of arrays is extended to multidimensional arrays. Declaration, initialization, usage and passing multidimensional arrays to functions will be treated both for arrays of numbers and arrays of strings.

All arrays discussed in chapters 7 and 8 are examples of one-dimensional arrays. C offers the possibility to use arrays with more than 1 dimension, called multidimensional arrays. Though, in principle, the number of dimensions is not limited, we will mainly discuss 2 and 3 D arrays.

## 9.1 Two dimensional arrays of numbers

A common use of two dimensional arrays is to represent tables of values arranged in rows and columns, also called matrices.

### 9.1.1 Declaration

In its most general form the declaration of a 2-dimensional array looks like:

```
type <name>[MAX_ROW][MAX_COLUMN];
```

where MAX_ROW / MAX_COLUMN indicate the number of rows / columns of the matrix.

ex: `int m[5][3];`

This declaration reserves memory for a 2-dimensional array with 5 rows and 3 columns. To address a particular table element, both a row and column index must be specified as illustrated in Figure 39.

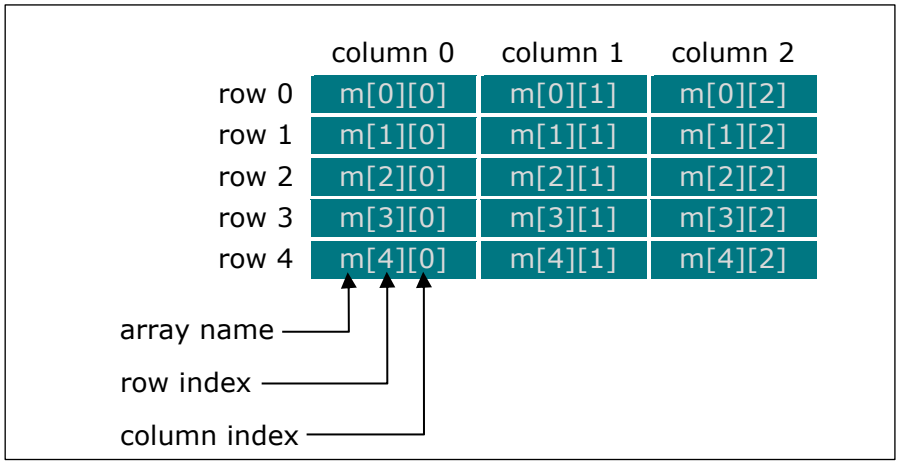



**Figure 39: logical structure of a 2-dimensional array**

In memory, this array is saved as 1 row of consecutive elements. First, all elements of row 0 are stored, then all elements of row 1, … For the array example of Figure 39, this is shown in Figure 40.

| m[0][0] | m[0][1] | m[0][2] | m[1][0] | m[1][1] | … | m[4][2] |
|---|---|---|---|---|---|---|

**Figure 40: physical structure of a 2-dimensional array**

### 9.1.2 Initialization

A 2-dimensional array can be initialized element by element or when it is defined. If all elements need to be initialized to the same value, a loop can be used (see section 9.1.3).

Examples:

```
int m[2][3];
m[0][0] = 0;
m[0][1] = 1;
m[0][2] = 2;
m[1][0] = 10;
m[1][1] = 11;
m[1][2] = 12;
```

```
int m[2][3] = { {0,1,2} , {10,11,12} };
```

The values are grouped by row. The first set of braces groups the values of row 0, the second set the values of row 1.

### 9.1.3 Matrix usage

To initialize, change, read the individual matrix elements, loops can be used. Since printing a matrix must be done row by row, most programmers will loop through the matrix on a row by row basis. Every row contains the same number of elements (equal to the number of columns), so addressing all row elements one by one can be done by looping over the different columns within that row. This is illustrated in the example of Code 43.

Example:
Write a program that reads the marks of 5 students for 2 different subjects and stores them in a 2-dimensional array.

Looping through the matrix can be illustrated in the flowchart of Figure 41.

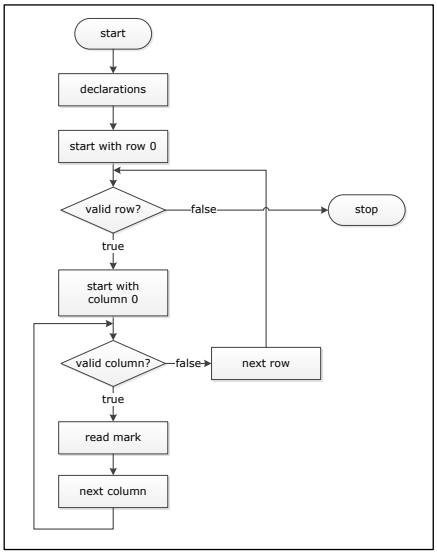



**Figure 41: flowchart looping through a matrix**

Translating into C code yields:

```
1   #include <stdio.h>
2   #define MAX_ROWS 5
3   #define MAX_COLUMNS 2
4
5   int main(void)
6   {
7      float marks[MAX_ROWS][MAX_COLUMNS];
8      int row, column;
9
10     //loop through rows
11     for (row = 0; row < MAX_ROWS; row++)
12     {
13            //loop through all columns in one row
14            for (column = 0; column < MAX_COLUMNS; column++)
15            {
16                   printf("Enter marks for student %d, subject %d: ",
                            row+1, column+1);
17                   scanf("%f%*c", &marks[row][column]);
18            }
19     }
20     return 0;
21  }
```

**Code 43: reading student marks by looping through a 2D array**

## 9.1.4 Passing a 2D array to a function

To pass a multidimensional array to a function, the array name is used as function argument just like for 1-dimensional arrays. As a result, the array starting address will be passed to the function and all modifications done on the matrix inside the called function, will be carried out on the original matrix.

However, there is one difference. As explained before, the length of a 1D array does not need to be specified in the function declaration and

definition. For multidimensional arrays all dimensions except the first one need to be specified!

This can be explained for the case of a 2-dimensional array. Remember that the array elements are stored in one long row, starting with all elements of row 0, followed by the elements of row 1, … If the matrix element `mat[i][j]` must to be accessed, the memory location needed can be calculated by:

```
starting address of the array + i * MAX_COLUMN + j
```

As clearly indicated by this formula, the first dimension (MAX_ROWS) is not needed to calculate the correct memory location whereas the second dimension is.

Example:
Add an extra function to the example of Code 43 to print the student marks as a 2-dimensional matrix.

```
1   #include <stdio.h>
2   #define MAX_ROWS 5
3   #define MAX_COLUMNS 2
4
5   void PrintMarks(float[][MAX_COLUMNS], int, int);
6
7   int main(void)
8   {
9      float marks[MAX_ROWS][MAX_COLUMNS];
10     int row, column;
11
12     for (row = 0; row < MAX_ROWS; row++)
13     {
14            for (column = 0; column < MAX_COLUMNS; column++)
15            {
16                   printf("Enter marks for student %d, subject %d: ",
                              row + 1, column + 1);
17                   scanf("%f%*c", &marks[row][column]);
18            }
19     }
20
21     PrintMarks(marks, MAX_ROWS, MAX_COLUMNS);
22
23     return 0;
24  }
25
26  void PrintMarks(float m[][MAX_COLUMNS], int maxrow, int maxcol)
27  {
28     int row, column;
29     printf("\n");
30
31     for (row = 0; row < maxrow; row++)
32     {
33            for (column = 0; column < maxcol; column++)
34            {
35                   printf("%5.2f\t", m[row][column]);
36            }
37            printf("\n");
38     }
39  }
```

```
C:\Users\u0088734\Documents\Tempus-DESIRE\C for embedded systems\chapter9\Debug\example...
Enter marks for student 1, subject 1: 10
Enter marks for student 1, subject 2: 11
Enter marks for student 2, subject 1: 5
Enter marks for student 2, subject 2: 9
Enter marks for student 3, subject 1: 12
Enter marks for student 3, subject 2: 18
Enter marks for student 4, subject 1: 5
Enter marks for student 4, subject 2: 12
Enter marks for student 5, subject 1: 13
Enter marks for student 5, subject 2: 13

10.00   11.00
 5.00    9.00
12.00   18.00
 5.00   12.00
13.00   13.00
```

**Code 44: passing a 2D array to a function**

### 9.1.5 2D array example: Pascal's triangle

Pascal's triangle is a triangular array containing the coefficients of the different terms of the expansion of $(a+b)^n$.

Each number in the triangle is the sum of the two numbers directly above. The first eight rows of Pascal's triangle are illustrated in Figure 42.

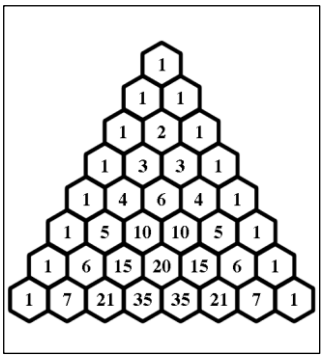
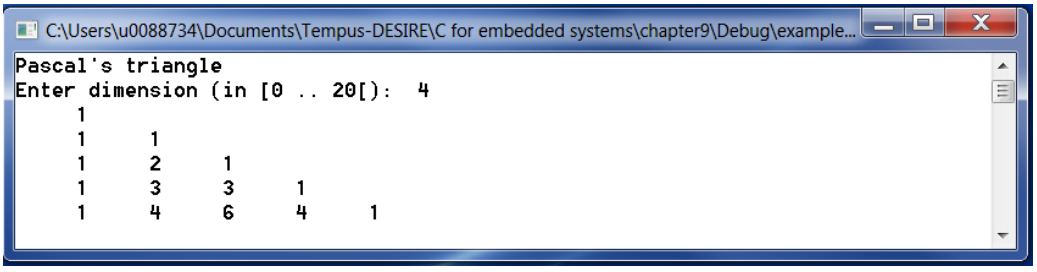



**Figure 42: Pascal's triangle**

Following code will print n rows of Pascal's triangle where n needs to be entered by the user:

```
1   /*
2      printing n rows of Pascal's triangle
3   */
4   #include <stdio.h>
5   #define MAX 20
6
7   int ReadDimension(void);
8   void MakeTriangle(int[][MAX], int);
9   void PrintTriangle(int[][MAX], int);
10
11  int main(void)
12  {
13     int a[MAX][MAX];
14     int n;
15
16     printf("Pascal's triangle\n");
17
18     n = ReadDimension();
19     MakeTriangle(a, n);
20     PrintTriangle(a, n);
21
22     return 0;
23  }
```

```
24  int ReadDimension(void)
25  {
26     int d;
27
28     do
29     {
30            printf("Enter dimension (in [0 .. %2d[):  ", MAX);
31            scanf("%d%*c", &d);
32     } while (d >= MAX);
33
34     return d;
35  }
36
37  void MakeTriangle(int a[][MAX], int n)
38  {
39     int i, j;
40
41     a[0][0] = 1;
42
43     for (i = 1; i <= n; ++i)
44     {
45            a[i][0] = 1;
46            a[i][i] = 1;
47     }
48
49     for (i = 2; i <= n; ++i)
50     {
51            for (j = 1; j<i; ++j)
52            {
53                   a[i][j] = a[i - 1][j - 1] + a[i - 1][j];
54            }
55     }
56  }
57
58  void PrintTriangle(int a[][MAX], int n)
59  {
60     int i, j;
61
62     for (i = 0; i <= n; ++i)
63     {
64            for (j = 0; j <= i; ++j)
65            {
66                   printf("%6d", a[i][j]);
67            }
68            printf("\n");
69     }
70  }
```




**Code 45: Pascal's triangle**

## 9.2 Arrays of strings

As explained before, a string is an array of characters. Therefore, an array of strings is a 2-dimensional array of characters where every row contains a string.

Figure 43 shows an array with name "`students`" for 5 strings of maximal 9 characters + 1 null byte.


**Figure 43: array of strings**

The name of the array "`students`" refers to the starting address of the full matrix. To access an individual character of one of the strings use both row and column index. For instance "`students[1][2]`" points to the symbol in row 1, column 2 (second row, third column).
To manipulate a full string at once, use a row index only. "`students[1]`" selects the string on row 1 (second row). This is a 1D array of characters, therefore "`students[1]`" points to the starting address of row 1.

Example:
Write a program that:
- reads a list of names and stores them in an array. The list of names ends with an empty string (enter only).
- reads an extra name
- checks if the extra name is present in the array or not

Following functions will be used:
- `ReadList`: reads the list of names and returns the number of names entered
- `ReadName`: reads 1 name
- `Find`: looks for 1 name in the array of names and returns the index number of that name in the array or -1 if the name was not found.

```
1   /*
2      find a name in a list of names
3   */
4
5   #include <stdio.h>
6   #include <string.h>
7   #include <stdlib.h>
8
9   #define MAX 50
10  #define LEN 20
```

```
11  int ReadList(char[][LEN]);
12  void ReadName(char[]);
13  int Find(char[][LEN], int, char[]);
14
15  int main(void)
16  {
17     char matrix[MAX][LEN];
18     char name[LEN];
19     int size, result;
20
21     // Read the list of names
22     size = ReadList(matrix);
23
24     // Read the name to be found
25     ReadName(name);
26
27     // Look for name in the list
28     result = Find(matrix, size, name);
29
30     if (result >= 0)
31           printf("The name \"%s\" was found at place %d.\n", name,
                    result+1);
32     else
33           printf("The name \"%s\" was not found.\n", name);
34
35     return 0;
36  }
37
38  int ReadList(char a[][LEN])
39  {
40     int i = 0;
41     char name[LEN];
42     printf("Enter name %2d (Enter = end) : ", i);
43     gets(name);
44
45     while (i < MAX && name[0] != '\0')
46     {
47           strcpy(a[i], name);
48           printf("Enter name %2d (Enter = end) : ", ++i);
49           gets(name);
50     }
51     if (i == MAX)
52     {
53           printf("Array is full!\n");
54           exit(1);
55     }
56     else
57     {
58           return i;
59     }
60  }
61
62  void ReadName(char x[])
63  {
64     printf("Enter the name you want to search for: ");
65     gets(x);
66  }
67
68  int Find(char a[][LEN], int n, char b[])
69  {
70     int i;
71     for (i = 0; i<n; ++i)
72     {
73           if (strncmp(a[i], b, LEN) == 0)
74                 return i;
75     }
76     return -1;
77  }
```

## 9.3 Exercises

**9.1.** Write a program with functions.
- in the `main` function, a table of 10 x 10 is defined
- the first function (`FillMatrix`) fills the matrix as shown below
- the second function (`PrintMatrix`) prints the content of the matrix.

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

**9.2.** Write a program with functions that first reads 2 x 10 integers and stores them on the first 2 rows of a matrix. In the third row, the sum of the corresponding numbers in the first 2 rows is stored. Finally, the full matrix is printed. Use the functions `ReadRow`, `Calculate` and `PrintMatrix`

```
Enter 2 x 10 integers:
1 2 3 4 5 6 7 8 9 10
1 2 2 1 3 4 1 1 2 1

Table:
   1   2   3   4   5   6   7   8   9  10
   1   2   2   1   3   4   1   1   2   1
   2   4   5   5   8  10   8   9  11  11
```

**9.3.** Write a program with functions that reads a list of integers and prints all integers of that list followed by the number of occurrences. The list is ended with the number 999 that cannot be taken into account. The list contains maximal 10 different numbers.

```
Enter a list of integers (end with 999):
-10 9 25 -10 100 25 25 3 0 25 100 3 999

The different numbers in this list are:
 -10   2
   9   1
  25   4
 100   2
   3   2
   0   1
```

**Hint**: use a 10 x 2 matrix. The first column contains the different integers and the second column the number of occurrences for the corresponding integer. Every time an integer is read, the program verifies if that number is already present or not. If already present, the number of occurrences is augmented, otherwise the number is added to the first free line with a number of occurrences equal to 1 in the second column.

**9.4.** Write a program that reads a square matrix, calculates its transpose and prints both the original and the transposed matrix. The matrix cannot be larger than 10 x 10. The wanted dimension is read at the start of the program. Reading the dimension, reading the matrix elements, transposing the matrix and printing the matrix is done in 4 different functions.

Remember:

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \end{bmatrix}^T = \begin{bmatrix} a_{0,0} & a_{1,0} & a_{2,0} \\ a_{0,1} & a_{1,1} & a_{2,1} \\ a_{0,2} & a_{1,2} & a_{2,2} \end{bmatrix}$$

**9.5.** Write a program with functions that:
- declares an array with dimensions 10 x 10 in the `main`
- uses a function to read the array elements
- uses a function to find the min and max element in the array and swaps them (inside the array, using an auxiliary variable is allowed)
- uses a function to print the array after swapping

**9.6.** Write a program with a `main` and 2 extra functions:
- declare an array of 10 strings in the `main` function
- the first function reads the 10 names and stores them in the array
- the second function prints the names in the array to the screen

**9.7.** Write a program with a `main` and 2 extra functions:
- declare an array of 10 strings in the `main` function
- the first function reads names until the word "end" or the maximum of 10 names is entered and stores them in the array
- the second function prints the names in the array to the screen

**9.8.** Write a program that reads a letter, converts it to Morse code and prints the result to the screen. You can use following array that contains the Morse codes for the letters A, B, C, … consecutively:

```
const char *morse[]={".-", "-...", "-.-.", "-..", ".", "..-
.", "--.", "....", "..",".---", "-.-", ".-..", "--", "-.", "-
--", ".--.", "--.-", ".-.", "...", "-", "..-", "...-", ".--",
"-..-", "-.--", "--..", NULL};
```

```
Enter a letter: c
Corresponding Morse code:  -.-.
```

**9.9.** Write a program that reads a word and prints it in Morse code to the screen. You can use the function from the previous exercise to write the Morse code letter by letter to the screen.

Enter a word: bread
Morse code: -... .-. . .- -..

**9.10.** Write a program to process test results.
To guarantee the product quality, a company takes a sample of N finished parts ($0 < N \leq 20$) and submits them to a series of M tests ($0 < M \leq 10$). If a part fails one or more tests, data is sent to the computer in the format:

```
PartNumber  TestNumber  Result
```

where $1 \leq$ `PartNumber` $\leq N$ and $1 \leq$ `TestNumber` $\leq M$
and `Result = 1` for a small error
`= 3` for a fatal error

A part is rejected if at least 1 fatal error or at least 3 small errors have occurred.

The program first reads the amount of parts tested (N) and the number of tests executed per part (M). Then, the test results for all failed parts are read until 0 0 0 is entered.
The program prints:

- a table with a line for every tested part (so also the ones that did not fail any test) containing information on the test results and a final assessment
- a second table with a line per test containing the number of parts that did not fail, the number of parts that showed a small error and the number of parts that showed a fatal error.

The program uses functions:

- a function `ReadNumber` that is used twice. Once to read the number of parts tested (N) and once to read the number of tests (M). Make sure only valid numbers are accepted.
- a function to read all test data
- a function to print the results per part
- a function to print the results per test
- optional: a function to count the number of occurrences of a certain value in a certain column

Hint: Use a matrix of 20 x 10 of which the upper left corner of N rows and M columns is used. Put every entry in the right place in that matrix. The first table to be printed is then just a printout of the used section of the matrix accompanied by some text. The second table can be constructed by counting the occurrences of the numbers 0, 1 and 3 in every column of the matrix.

The screen dialog should look like:

```
How many parts did you test? 5

How many tests did you run? 3

Enter the test results (end with 0 0 0):
4 1 1
5 3 3
3 2 3
2 3 1
3 3 1
4 3 1
4 2 1
0 0 0

Results per part:
part number    tests     assessment
               1  2  3
-----------------------------------
1              0  0  0     accepted
2              0  0  1     accepted
3              0  3  1     rejected
4              1  1  1     rejected
5              0  0  3     rejected

Results per test:
test          failures
        none  small fatal
---------------------------
1       4     1     0
2       3     1     1
3       1     3     1
```

**9.11.** Write a program that simulates seat reservations in a theater. Let's consider a room with 3 rows of 4 seats each. Define a matrix of 3 x 4 that represents the room. Every matrix element must be large enough to contain a string of maximal 10 symbols, resulting in a 2D array of strings which is a 3D array of characters! The program repeats itself until the user enters 0 0 0 or until all seats are taken.
In every program run, a name, row number and seat number are read. If the seat is still available, the name is stored in the corresponding row and column of the matrix. If the seat was already taken, the program prints "occupied". After every seat reservation, all reservations are printed to the screen.

Use functions to write the program:

- declare the room in the `main` program
- write a function to print the current reservations
- write a function to initialize the reservations
- write a function to read the inputs and check the availability of the wanted seat

```
Welcome. The theater has 3 rows of 4 seats each.
The current reservations are:
    .      .      .      .
    .      .      .      .
    .      .      .      .

Enter a new seat reservation: Mary 1 2
The current reservations are:
    .      Mary   .      .
    .      .      .      .
    .      .      .      .

Enter a new seat reservation: Carl 3 3
The current reservations are:
    .      Mary   .      .
    .      .      .      .
    .      .     Carl    .

Enter a new seat reservation: end
The final seat reservations are:
    .      Mary   .      .
    .      .      .      .
    .      .     Carl    .
```

# 10 Sorting and searching arrays

**Objectives**

In this chapter we will look into different methods to sort arrays and search elements in an array.

## 10.1 Sorting arrays of numbers

In this example, we will use following functions:

- `ReadSize`: to read how many integers will need to be ordered
- `ReadArray`: to read the set of integers
- `SortArray`: orders the integers ascending
- `PrintArray`: prints the ordered integers
- `FindIndexSmallest`: searches the index belonging to the smallest number

The flowchart of Figure 44 illustrates the algorithm used for the functions `SortArray` and `FindIndexSmallest`:

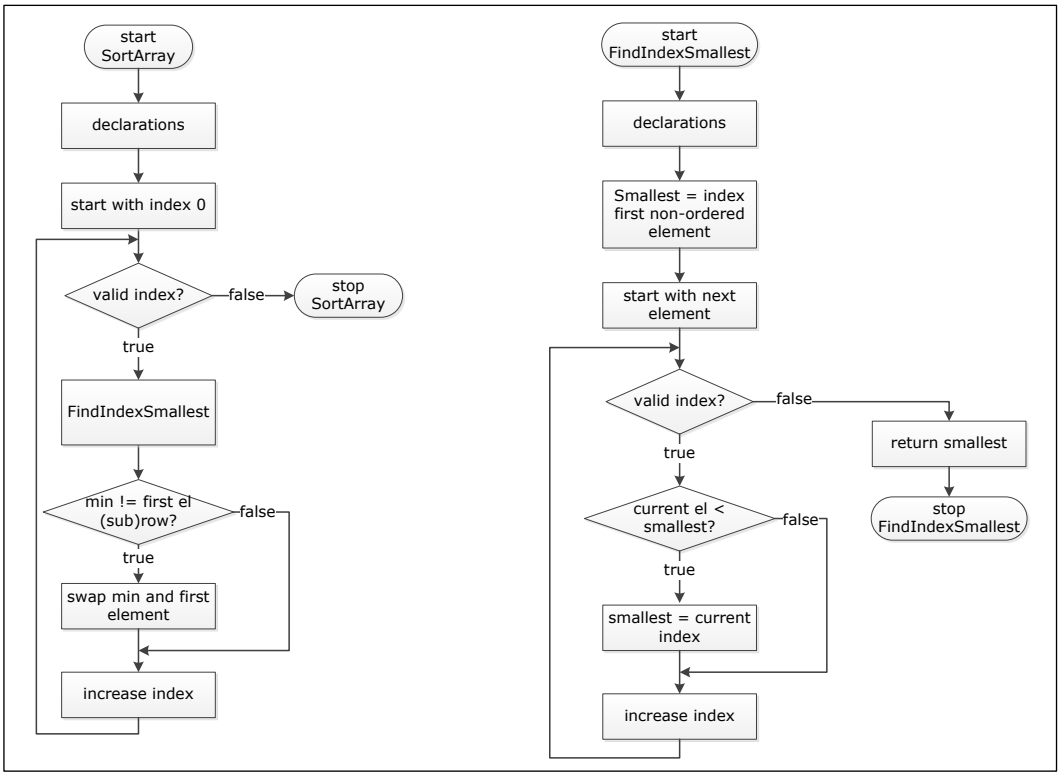



**Figure 44: flowchart sorting algorithm numbers**

The full C program can be written as follows:

```
1   /*
2      sorting an array of integer numbers
3   */
4
5   #include <stdio.h>
6
7   #define MAX_SIZE 10
8
9   int ReadSize(void);
10  void ReadArray(int[], int);
11  void SortArray(int[], int);
12  int FindIndexSmallest(int[], int, int);
13  void PrintArray(int[], int);
14
15  int main(void)
16  {
17     int row[MAX_SIZE];
18     int size;
19
20     printf("Sorting a list of integers\n");
21
22     // read number of integers that will be entered
23     size = ReadSize();
24
25     // read integers
26     ReadArray(row, size);
27
28     // order row
29     SortArray(row, size);
30
31     // print ordered row
32     PrintArray(row, size);
33
34     return 0;
35  }
36
37  int ReadSize(void)
38  {
39     int n;
40
41     do
42     {
43           printf("Enter the number of integers you want to
                     sort:[1..%d] ", MAX_SIZE);
44           scanf("%d%*c", &n);
45     } while (n < 1 || n > MAX_SIZE);
46
47     return n;
48  }
49
50
51  void ReadArray(int a[], int n)
52  {
53     int i;
54
55     printf("Enter %d integers:\n", n);
56
57     for (i = 0; i < n; ++i)
58           scanf("%d%*c", &a[i]);
59  }
60
```

```
61  void SortArray(int r[], int n)
62  {
63     int i, temp, IndexSmallest;
64
65     for (i = 0; i < n - 1; ++i)
66     {
67            // find index smallest number  in (sub)row
68            IndexSmallest = FindIndexSmallest(r, n, i);
69
70            // if smallest element != first element of (sub)row,
                 switch first and smallest
71            if (r[IndexSmallest] != r[i])
72            {
73                   temp = r[IndexSmallest];
74                   r[IndexSmallest] = r[i];
75                   r[i] = temp;
76            }
77     }
78  }
79
80  int FindIndexSmallest(int a[], int n, int start)
81  {
82     int i, index_s;
83
84     index_s = start;
85
86     for (i = start + 1; i < n; ++i)
87            if (a[i] < a[index_s])
88                   index_s = i;
89
90     return index_s;
91  }
92
93  void PrintArray(int x[], int a)
94  {
95     int j;
96
97     printf("The ordered row:\n");
98
99     for (j = 0; j<a; ++j)
100           printf("%8d", x[j]);
101
102    printf("\n");
103 }
```

**Code 46: sorting arrays of numbers**

## 10.2 Sorting arrays of strings

We will use the same sorting algorithm as in section 10.1. The only difference is that we will use the string function `strcmp` to alphabetize the names.

```
1   /*
2      sorting an array of names
3   */
4   #include <stdio.h>
5
6   #define MAX_SIZE 10
7   #define MAXLENGTH 32
8
9   int ReadSize(void);
10  void ReadArray(char[][MAXLENGTH], int);
11  void SortArray(char[][MAXLENGTH], int);
```

```
12  int FindIndexSmallest(char[][MAXLENGTH], int, int);
13  void PrintArray(char[][MAXLENGTH], int);
14
15  int main(void)
16  {
17     char row[MAX_SIZE][MAXLENGTH];
18     int size;
19
20     printf("Sorting a list of names\n");
21     size = ReadSize();
22     ReadArray(row, size);
23     SortArray(row, size);
24     PrintArray(row, size);
25
26     return 0;
27  }
28
29  int ReadSize(void)
30  {
31     int n;
32
33     do
34     {
35           printf("Enter the number of names you want to
                    sort:[1..%d] ", MAX_SIZE);
36           scanf("%d%*c", &n);
37     } while (n < 1 || n > MAX_SIZE);
38
39     return n;
40  }
41
42  void ReadArray(char a[][MAXLENGTH], int n)
43  {
44     int i;
45
46     printf("Enter %d names:\n", n);
47
48     for (i = 0; i < n; ++i)
49           gets(a[i]);
50  }
51
52  void SortArray(char r[][MAXLENGTH], int n)
53  {
54     int i, IndexSmallest;
55     char temp[MAXLENGTH];
56
57     for (i = 0; i < n - 1; ++i)
58     {
59           IndexSmallest = FindIndexSmallest(r, n, i);
60           if (strcmp(r[IndexSmallest], r[i]))
61           {
62                 strcpy(temp, r[IndexSmallest]);
63                 strcpy(r[IndexSmallest], r[i]);
64                 strcpy(r[i], temp);
65           }
66     }
67  }
68
69  int FindIndexSmallest(char a[][MAXLENGTH], int n, int start)
70  {
71     int i, index_s;
72     index_s = start;
73     for (i = start + 1; i < n; ++i)
74           if (strcmp(a[i], a[index_s]) < 0)
75                 index_s = i;
76     return index_s;
77  }
```

```
78  void PrintArray(char x[][MAXLENGTH], int a)
79  {
80     int j;
81
82     printf("The ordered row:\n");
83
84     for (j = 0; j < a; ++j)
85           printf("%s\n", x[j]);
86
87     printf("\n");
88  }
```

**Code 47: sorting arrays of names**

## 10.3 Binary search

Searching an array can be done in different ways. The most straight forward method is the one we have used so far: simply checking the elements of the array one by one from the first element until the last one. This is called linear searching.

If for instance a number needs to be found in an ordered row, a binary search algorithm will be much more efficient. In this case, the number to be found is compared to the middle number of the array. If both are equal, the number is found. If not, we will go on searching but only in the first or second half of the original row. So with one check, we can immediately eliminate half of the row. The search continues according to the same principle until the number is found or the remaining part of the row is empty.

This algorithm of the binary search is represented in Figure 45:

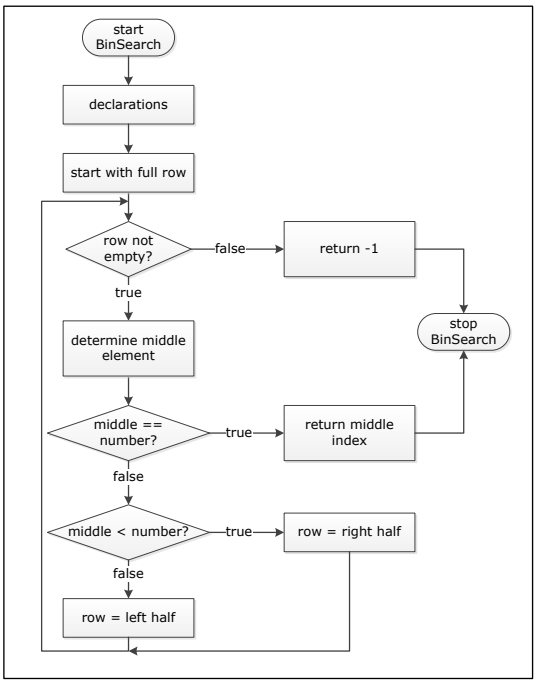



**Figure 45: flowchart binary search algorithm**

The full C program can be written as shown in Code 48.

```
1   /*
2      binary search in an ordered row of integers.
3   */
4
5   #include <stdio.h>
6   #include <stdlib.h>
7
8   #define MAX_SIZE 10
9
10  int ReadSize(void);
11  void ReadArray(int[], int);
12  int ReadNumber(void);
13  int BinSearch(int[], int, int);
14
15  int main(void)
16  {
17     int a[MAX_SIZE];
18     int size, number, index;
19
20     // number of integers you want to read ( < MAX_SIZE )
21     size = ReadSize();
22
23     // read the array of integers
24     ReadArray(a, size);
25
26     // read the number to search for
27     number = ReadNumber();
28
29     // find number in row + determine position
30     index = BinSearch(a, size, number);
31
32     if (index >= 0 && index < size)
33           printf("The number is present on place %d\n", index);
34     else
35           printf("The number is not present.\n");
36
37     return 0;
38  }
39
40
41  int ReadSize(void)
42  {
43     int n;
44
45     do
46     {
47           printf("How many integers do you want to
                    enter [1..%2d]: ", MAX_SIZE);
48           scanf("%d%*c", &n);
49     } while (n < 1 || n > MAX_SIZE);
50
51     return n;
52  }
53
54  int ReadNumber(void)
55  {
56     int n;
57
58     printf("Enter the number to search for: ");
59     scanf("%d%*c", &n);
60
61     return n;
62  }
```

```
63  void ReadArray(int x[], int a)
64  {
65     int i;
66
67     printf("Enter the %d integers, ordered from smallest to
                largest:\n", a);
68
69     for (i = 0; i < a; ++i)
70     {
71            scanf("%d%*c", &x[i]);
72
73            // test if well ordered
74            if (i > 0 && x[i] < x[i - 1])
75            {
76                   printf("The ordering is not correct\n");
77                   exit(5);
78            }
79     }
80  }
81
82  int BinSearch(int a[], int n, int g)
83  {
84     int first, last, middle;
85
86     first = 0;
87     last = n - 1;
88
89     while (last >= first)
90     {
91            // integer division!!
92            middle = (first + last) / 2;
93
94            // determine new subarray if number not found yet
95            if (a[middle] == g)
96                   return middle;
97            else if (a[middle] < g)
98                   first = middle + 1;
99            else
100                  last = middle - 1;
101    }
102    return -1;
103 }
```

**Code 48: binary search**

## 10.4 Exercises

**10.1.** Write a program with functions.
- In the `main` function, an array of 10 integers is declared.
- A first function reads the 10 integers and stores them in the array.
- A second function swaps the first integer with the min value present in the array. Swapping must be done in the same array, without using a second array. Using an extra auxiliary variable is allowed. All other numbers need to stay in their original places.
- A third function prints the array after swapping.

**10.2.** Bubble sort is another method used to order arrays. It is an algorithm that repeatedly steps through the list to be sorted. In each pass, each pair of adjacent items is compared and swapped if they are in the wrong order. If, for instance, ordering from smallest to largest is needed, the biggest element will end up at the end of the list after the first pass. The algorithm, is named for the way bigger elements "bubble" to the end of the list. The next pass can stop a bit sooner (the last element is already in place). How many passes are needed? Can you stop sooner?
Write a program, with functions, that orders a list of numbers using the bubble sort method.

**10.3.** Use bubble sort to order a list of names.

**10.4.** Write a program with (minimal) 4 functions that:
- reads a list of 9 integers. The integers must be entered in ascending order. Make sure your program verifies if the numbers entered are correctly ordered. If not, the program prints a message and stops execution.
- reads a $10^{th}$ integer.
- finds the correct location for this $10^{th}$ integer within the sorted list and inserts it there.
- prints the final list.

**10.5.** Write a sorting program based upon exercise 10.4.
- read an integer.
- read a next integer and put it in the correct place in the list.
- repeat step 2 until all numbers are read.

This method is called insertion sort.

# 11 Pointers

**Objectives**

In this chapter the basic concepts of pointers are explained. At the end of this chapter, one should be able to:
- declare and initialize a pointer
- use pointers and pointer operators
- understand the close relationship between pointers and arrays
- use pointers to pass arguments by reference to a function
- use pointers to functions

Pointers are an extremely powerful programming tool. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. Pointers are used, for example, to have a function directly modify a variable passed to it. It is also possible to use pointers to dynamically allocate memory, which means that you can write programs that can handle big amounts of data on the fly. You don't need to know, when you write the program, how much memory you need.

This chapter will treat the basic concepts of pointers. Using pointers for dynamic memory allocations will be treated in chapter 17.

## 11.1 Definition

In C, every variable has a type, an address in memory and a value. A pointer is a variable whose value is the address of another variable. Hence, you can say that the pointer "points" to that other variable. This is illustrated in Figure 46:

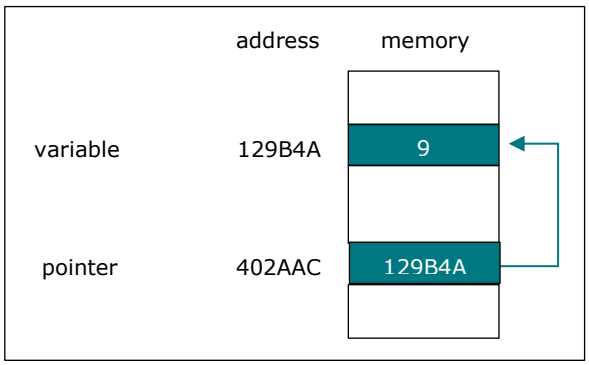



**Figure 46: pointer principle**

## 11.2 Declaration and initialization

### 11.2.1 Declaration

Like every variable in C, a pointer variable must be declared before it can be used. In its most general form, a pointer declaration can be written as:

```
<target_datatype> * <name>;
```

The `*` in the declaration indicates that the variable is a pointer. For instance

```
int *ptr;
```

declares the pointer variable with name `ptr` that can point to a variable of the type `int`.

**Remark**

When a pointer is declared, memory is reserved to store an address only! No memory is reserved for the variable the pointer will refer to, nor does the pointer point to any variable yet!

### 11.2.2 Initialization

Pointers should be initialized when defined or they should be assigned a value. Pointers can be initialized to NULL or an address. A pointer initialized to NULL is a pointer that points to nothing.

**Learning note**

To avoid unexpected behavior, always initialize pointers!

## 11.3 Address and dereference operator

Now that the pointer is declared and initialized, we still need to make sure it points to the variable of our choice. To this end, the address of that variable must be assigned to the pointer.

To determine the memory address of a variable, the address operator `&` is used:

```
int *p = NULL;      //declaration and initialization of the pointer
int a = 5;          //declaration of the integer variable a

p = &a;
```

The statement "`p = &a;`" assigns the address of the integer variable `a` to the pointer `p`. As a result, the pointer `p` now points to the variable `a`.

On the other hand, it must also be possible to read the value of the variable the pointer is pointing to. The operator to be used is called the dereferencing operator and is represented by the symbol *.

```
int *p = NULL;      //declaration and initialization of the pointer
int a = 5;          //declaration of the integer variable a
p = &a;

printf("%d", *p);
```

The statement "`printf("%d", *p);`" will output the number 5 to the screen. `*p` performs the dereferencing operation on the pointer `p`. It reads the address stored in the pointer variable `p`, goes to that memory location and returns the value stored at that location.

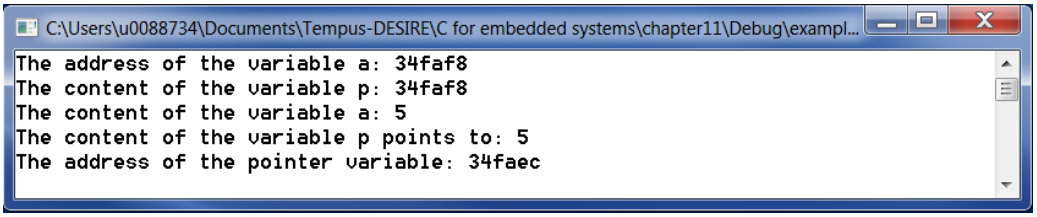
The same operator can be used to store values into the variable the pointer refers to:

```
*p = 8;
```

When above statement is executed, the address stored in the pointer variable `p` will be read and the number 8 will be stored at that memory location. Therefore, in the above example, the variable `a` will get the value 8.

The example of Code 49 demonstrates the address and the dereferencing operator:

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5      int a = 5;
6      int *p = NULL;
7      p = &a;
8      printf("The address of the variable a: %x\n", &a);
9      printf("The content of the variable p: %x\n", p);
10     printf("The content of the variable a: %d\n", a);
11     printf("The content of the variable p points to: %d\n", *p);
12     printf("The address of the pointer variable: %x\n\n", &p);
13     return 0;
14  }
```



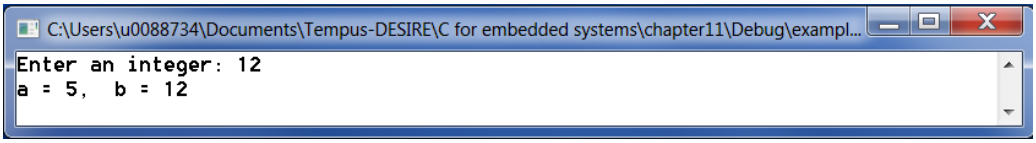

**Code 49: demonstration of & and * operators**

**Common mistake**

Dereferencing a pointer that was not initialized nor assigned to a specific memory location is a commonly made mistake.

The following pointer example shows how pointers can be changed to point to different variables in 1 program. Of course, the pointer always points to one variable at a time!

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5      int a, b;
6      int *p = NULL;
7
8      p = &a;
9      *p = 5;
10
11     p = &b;
12     printf("Enter an integer: ");
13     scanf("%d%*c", p);
14
15     printf("a = %d,\tb = %d\n", a, b);
16
17     return 0;
18  }
```




**Code 50: pointer usage**

Note that the `scanf` function gets `p` as argument. The argument must indicate the memory address where the integer can be stored. This address is stored in the pointer variable `p`.

**Common mistake**

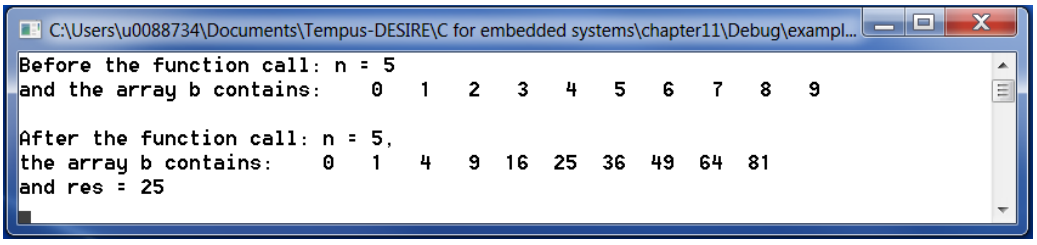
Using an address operator before a pointer variable in a `scanf` function call results in unexpected behavior of the program.

## 11.4 Passing arguments to functions

### 11.4.1 Pass by value

If a function with parameters is called, the values of the arguments are copied to the function parameters. Inside the function, these parameters can change value. Once the function is finalized, the changes done to the function parameters are not automatically copied to the original variables used as arguments. If one of the changed values needs to be copied to the calling function, a return statement must be used. This behavior is illustrated in the example of Code 51.

```
1   #include <stdio.h>
2   #define SIZE 10
3
4   int function(int, int[]);
5
6   int main(void)
7   {
8      int i, n, res;
9      int b[SIZE];
10
11     n = 5;
12     for (i = 0; i < SIZE; ++i)
13           b[i] = i;
14
15     printf("Before the function call: n = %d\n", n);
16     printf("and the array b contains: ");
17     for (i = 0; i < SIZE; ++i)
18           printf("%4d", b[i]);
19
20     res = function(n, b);
21
22     printf("\n\nAfter the function call: n = %d,\n", n);
23     printf("the array b contains: ");
24     for (i = 0; i < SIZE; ++i)
25           printf("%4d", b[i]);
26
27     printf("\nand res = %d\n", res);
28
29     return 0;
30  }
31
32  int function(int x, int y[])
33  {
34     int i;
35
36     x = x * x;
37
38     for (i = 0; i < SIZE; ++i)
39           y[i] = i * i;
40
41     return x;
42  }
```




**Code 51: pass by value**

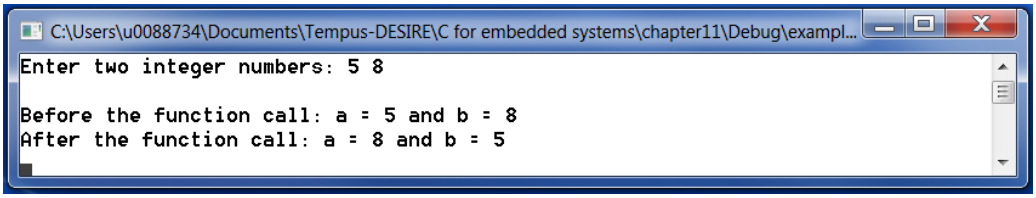
In this example, the value of `n` is copied into the function parameter `x`. In the function, `x` is modified, but the variable `n` in the main function remains unchanged. The array `b` on the other hand is changed from within the function. In the case of an array, the starting address of that array is passed to the function. As a result the array in the function (`y`) and the original array (`b`) point to the same memory location!

### 11.4.2 Pass by reference

Often, the changes carried out on the parameters in the called function need to affect also the arguments in the calling function. To this end, a mechanism similar to passing arrays to functions needs to be used. Instead of passing the value of a variable to the function, the address of that variable needs to be passed to the function parameter. As a result, the function parameter is now a pointer that points to the original variable. So, through the pointer in the called function, the original variable can be altered.

Code 52 shows an example with a `swap` and a `read` function that use pass by reference.

```
1   #include <stdio.h>
2
3   void read(int *, int *);
4   void swap(int *, int *);
5
6   int main(void)
7   {
8      int a, b;
9
10     read(&a, &b);
11
12     printf("\nBefore the function call: a = %d and b = %d\n", a, b);
13     swap(&a, &b);
14     printf("After the function call: a = %d and b = %d\n", a, b);
15
16     return 0;
17  }
18
19  void read(int *x, int *y)
20  {
21     printf("Enter two integer numbers: ");
22     scanf("%d%d%*c", x, y);
23  }
24
25  void swap(int *x, int *y)
26  {
27     int temp;
28
29     temp = *x;
30     *x = *y;
31     *y = temp;
32  }
```




**Code 52: pass by reference**

The pointers `x` and `y` in the functions `swap` and `read` point to the original variables `a` and `b`. Therefore, `a` and `b` can be modified directly from inside the functions.

## 11.5 Pointers and arrays

Pointers and arrays are closely related in C. For instance, pointers can be used to point to an element in an array.

Consider following array and pointer declarations:

```
int arr[5] = {1, 2, 3, 4, 5};
int *arrPtr = NULL;
```

The name of an array evaluates to the starting address of the array. Therefore, an array name can be considered as a pointer to the first array element. To make sure the pointer `arrPtr` points to the first element of the array `arr` the following statement can be used:

```
arrPtr = arr;
```

or alternatively:

```
arrPtr = &arr[0];
```

At this point, the pointer is set as illustrated in Figure 47.

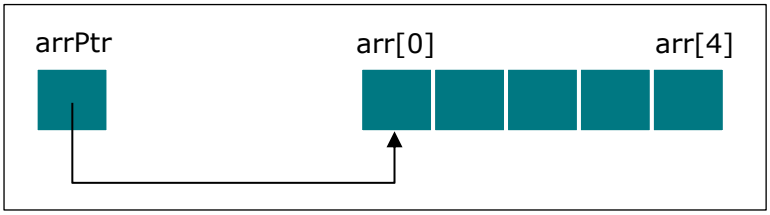



**Figure 47: pointer to array**

The value of the array element `arr[2]` can be accessed with the pointer expression:

```
*(arrPtr + 2);
```

`arrPtr` contains the starting address of the array. Adding an offset of 2 to that starting address, brings us to the element with index 2 inside the array.

Alternatively, since the array name and the pointer both contain the same memory address, we can use the array notation in combination with the pointer name, resulting in:

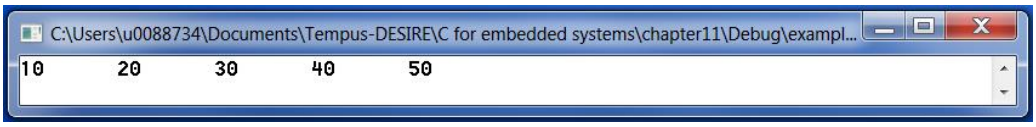
```
arrPtr[2];
```

Or, we could move the pointer such that it points directly to the array element with index 2:

```
arrPtr += 2;
```

Example:

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5      int numbers[5], *point;
6
7      point = numbers;
8     *point = 10;
9
10     point++;
11    *point = 20;
12
13     point = &numbers[2];
14    *point = 30;
15
16     point = numbers + 3;
17    *point = 40;
18
19     point = numbers;
20    *(point+4) = 50;
21
22     for(int n=0; n<5; n++)
23            printf("%d\t",numbers[n]);
24
25     return 0;
26  }
```




**Code 53: pointers and arrays**

## 11.6 Pointer versions of some string functions

To illustrate the usage of pointers, some string functions are rewritten.

### 11.6.1 strlen

```
1   int strlen(char *s)
2   {
3       char *p = s;
4       while( *p != '\0')
5           p++;
6       return p - s;
7   }
```

### 11.6.2 strcpy

```
1   void strcpy (char *s, char *t)
2   {
3       while( (*s = *t) != '\0')
4       {
5           s++;
6           t++;
7       }
8   }
```

### 11.6.3 strcmp

```
1   int strcmp (char *s, char *t)
2   {
3       for ( ; *s == *t; ++s, ++t)
4       {
5           if (*s == '\0')
6               return 0;
7       }
8       return *s - *t;
9   }
```

## 11.7 Pointers to functions

### 11.7.1 Function pointers

Like variables, functions need to be stored in memory. Therefore, also functions have a starting address. In C, it is possible to declare a pointer that points to the starting address of a function.

As an example we will write a program with a `main` and two functions where one of the functions is called by the use of a function pointer.

```
1   #include <stdio.h>
2
3   float OneThird(float);
4   float OneFifth(float);
5
6   int main()
7   {
8      float(*pf)(float);
9      pf = OneThird;
10     printf("%f\n", (*pf)(3.0));
11     return 0;
12  }
13
14  float OneThird(float x)
15  {
16     return x / 3;
17  }
18
19  float OneFifth(float x)
20  {
21     return x / 5;
22  }
```

**Code 54: function pointer**

The line

```
float (*pf) (float);
```

declares a variable `pf` that is a pointer (`*pf`) to a function that receives a `float` as parameter and returns a `float`. The parentheses around `*pf` are needed to indicate that `pf` is a pointer.

Without those parentheses, the instruction would become:

```
float * pf (float);
```

which declares a function with name `pf` that receives a `float` as argument and has a pointer to a `float` as return value.

The next statement:

```
pf = OneThird;
```

assigns the starting address of the function with name `OneThird` to the pointer `pf`. Like arrays, the name of a function refers to the starting address of that function in memory.

To use the function the pointer is pointing to, the pointer needs to be dereferenced. On top the correct function arguments need to be passed:

```
printf("%f\n", (*pf)(3.0));
```

### 11.7.2 Array of function pointers

Function pointers are commonly used in programs where a user can select a function to be carried out from a list of options. Using an array of function pointers to all possible functions, the user's choice can be translated into an index that allows to select the correct function pointer from the array. This is illustrated in the next example:

```
1   #include <stdio.h>
2
3   int add(int, int);
4   int subtract(int, int);
5
6   int main(void)
7   {
8      int a, choice;
9      int(*fptr[2])(int, int) = { add, subtract };
10
11     printf("Enter your choice:\n");
12     printf("\t0:\taddition (10 + 2)\n\t1:\tsubtraction (10 - 2)\n");
13     scanf("%d%*c", &choice);
14
15     a=(*fptr[choice])( 10, 2);
16
17     printf("The requested operation gives: %d\n", a);
18     return 0;
19  }
20
21  int add(int x, int y)
22  {
23     return x + y;
24  }
25
26  int subtract(int x, int y)
27  {
28     return x - y;
29  }
```

**Code 55: array of function pointers**

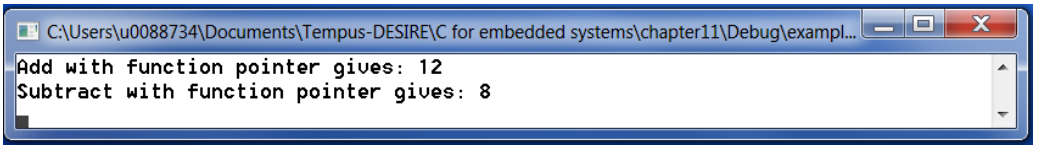
If the users chooses option 0 (addition), the element with index 0 is selected in the array, resulting in a function pointer to the function with name `add`.

### 11.7.3 Function pointers as function argument

Finally, function pointers allow using functions as parameters of other functions as illustrated in the examples of Code 56 and Code 57.

Example:
add and subtract with a function that receives a function pointer as argument

```
1   #include <stdio.h>
2
3   int add(int, int);
4   int subtract(int, int);
5   int domath(int(*)(int, int), int, int);
6
7   int main(void)
8   {
9      int a, b;
10
11     a = domath(add, 10, 2);
12     printf("Add with function pointer gives: %d\n", a);
13
14     b = domath(subtract, 10, 2);
15     printf("Subtract with function pointer gives: %d\n", b);
16
17     return 0;
18  }
19
20  int add(int x, int y) {
21     return x + y;
22  }
23
24  int subtract(int x, int y) {
25     return x - y;
26  }
27
28  // run the function pointer with inputs
29  int domath(int(*mathop)(int, int), int x, int y) {
30     return (*mathop)(x, y);
31  }
```




**Code 56: function pointer as function argument (add - subtract)**

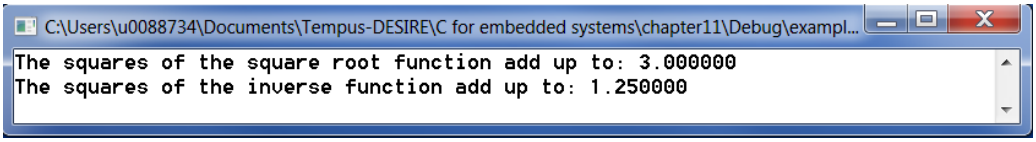
The first argument of the function `domath` is a function pointer with name `mathop` that can point to a function that receives two integers as inputs and returns an integer. Calling the function `domath` with `add` as first parameter, assigns the function `add` to the function pointer `mathop`.

Example:
calculate $\sum_{k=m}^{n} f^2(k)$ where f(k) can be chosen from:

$f(k) = sin(k)$

$f(k) = \frac{1}{k}$

```
1   #include <stdio.h>
2   #include <math.h>
3
4   double inverse(double);
5   double SumOfSquare(double(*)(double), double, double);
6
7   int main(void)
8   {
9      double start, end, res;
10     start = 1;
11     end = 3;
12
13     res = SumOfSquare(sqrt, start, end);
14
15     printf("The squares of the square root function
                add up to: %lf\n", res);
16
17     res = SumOfSquare(inverse, start, end);
18
19     printf("The squares of the inverse function
                add up to: %lf\n\n", res);
20     return 0;
21  }
22
23  double inverse(double x)
24  {
25     return 1.0 / x;
26  }
27
28  double SumOfSquare(double(*fptr)(double), double m, double n)
29  {
30     int k;
31     double sum = 0;
32     for (k = m; k < n; k++)
33     {
34            sum += (*fptr)(k)* (*fptr)(k);
35     }
36     return sum;
37  }
```




**Code 57: function pointer as function argument (sum of squares)**

## 11.8 Exercises

**11.1.** Write a program without arrays that:
- declares 3 variables of the type `double` in the `main` function
- reads all 3 variables with one function call to the function `ReadValues`
- prints the values of the 3 variables in the `main` function

**11.2.** Write a program to swap two numbers. The first number is stored in the variable `number1` and the second in the variable `number2`. After the swap the variable `number1` contains the second number and `number2` contains the first number. Use functions: `Read`, `Swap`, `Print`. Do not use arrays!

```
Enter 2 numbers: 5 10
The value of the first variable is 5, the value of the
second variable is 10.
After the swap, variable 1 contains 10 and variable 2
contains 5.
```

**11.3.** Write a program that converts a number of seconds into a number of hours, minutes and seconds. Reading the number of seconds and printing the result is done in the `main` function. The calculations are done in a separate function. No arrays are to be used!

```
Enter a time in seconds: 10000
The entered time of 10000 seconds equals 2 hours,
46 minutes and 40 seconds.
```

**11.4.** Write a program that reads an amount of money (≤ € 200) and that determines the minimum number of notes and coins needed to obtain that amount. Write a function for the calculations and a function to print to the screen. Do not use arrays!

```
Enter an amount of money: 195
195 euro can be obtained with:
     1  hundred
     1  fifty
     2  twenty
     1  five
     0  one
```

**11.5.** Write a program that reads the temperatures of a full week, finds the min and max temperatures and prints them together with all entered temperatures.

print the temperatures as follows:

```
       Mon   Tue   Wed   Thu   Fri   Sat   Sun
Day    5.0   5.5   7.0   6.0   7.5   8.0   8.5
Night -1.5  -0.5   0.0  -1.0   0.0   1.0   1.5
```

Print also min and max temperatures of day and night together with the day that temperature was measured:

```
Min:  Night:      Mon = -1.5
      Day:        Mon =  5.0
Max:  Night:      Sun =  1.5
      Day:        Sun =  8.5
```

The temperatures are stored in a 2D array with row 0 for the temperatures measured during the day and row 1 for the temperatures measured during the night. The names of the days are stored in a separate array (you cannot store numbers and text in the same array).

Use 3 functions: `Read`, `Print` and `Calculate`. Pass the indexes of the min and max temperatures for day and night by reference.

**11.6.** Write the function `print_string` to finalize the next program. The function `print_string` prints a string, received as argument, character by character using the function `putchar()`.

```
#include <stdio.h>
#include <string.h>
#define MAXSTRING 100

void print_string(char *c);

int main(void)
{
    char s1[MAXSTRING], s2[MAXSTRING];

    strcpy(s1, "Mary, Mary, quite contrary.\n");
    strcpy(s2, "How does your garden grow?\n");

    print_string(s1);
    print_string(s2);
    strcat(s1, s2);
    print_string(s1);
}
```

**11.7.** Write a program that reads a word and converts it as follows:

- the first 2 characters are printed.
  Ex: "qwerty" => print qw
- the first and second character are compared. The largest one is printed. Ex: q < w => print w as third letter
- now, compare the second and third character and again print the largest. Ex: w > e => print w as forth letter
- repeat until the word is finished

Once the conversion is finished, the program prints also the alphabetically smallest and largest letter.

The function `main` contains:

- asking the question "again?" and reading a 0 or 1 as answer
- printing the largest and smallest letter

The function `Read` takes care of:

- reading a word. Make sure only words of 2 or more characters can be entered.

The function `Print` takes care of:

- determining the smallest and largest letter and passing them to the `main` program with pointers
- converting and printing the word

```
Enter a word or a series of letters:
qwerty

qwwwrty
largest = y and smallest = e

again? (1=yes, 0=no): 1
Enter a word or a series of letters:
beverage

beevvrrgg
largest = v and smallest = a

again? (1=yes, 0=no): 0
```

**11.8.** Write a program that prints the tables of multiplication of an integer number entered by the user up to a limit that is also entered by the user. Keep on repeating the program until 0 0 is entered.
Use following functions:

- `Read`: read 2 integers (number and limit)
- `CalcPrint`: prints the table of multiplication and calculates the sum of the odd and even numbers
- `main`: calls the functions `Read` and `CalcPrint` and prints the sum of the odd and even numbers

No arrays can be used!

```
enter the number you want to use for the table of
multiplication:
5
enter the limit:
32

5
10
15
20
25
30

the sum of the even numbers is 60
the sum of the odd numbers is 45

enter the number you want to use for the table of
multiplication:
0
enter the limit:
0
Thanks!
```

**11.9.** Write a program that performs a mathematical computation on 2 numbers entered by the user.
The numbers are read in the function `Read` (do not use arrays!)

The `main` function:

- calls the function `Read`
- asks the user to choose an operator. Do this with a menu. Make sure only valid inputs are allowed.
- uses a function pointer that points to the correct function (use a `switch` statement to select the correct function)
- prints the result

```
Enter 2 integer numbers: 10 5

Choose an operand:
0   addition
1   subtraction
2   multiplication
3   division
4

Choose an operand:
0   addition
1   subtraction
2   multiplication
3   division
0
The result of this operation is: 15
```

**11.10.** Repeat the previous exercise but this time:

- make an array of function pointers where the function pointer stored at index 0 points to addition, at index 1 to subtraction, …
- write a separate function "`Choose`" that returns the user's choice to the main program
- make sure the integer returned by the function `Choose` is equal to the index corresponding to the correct function pointer.

# 12 Comma operator, const, typedef, enumerations and bit operations

**Objectives**

In this chapter some smaller topics often used in C programming are grouped together. You will learn about:
- the comma operator
- using the type qualifier `const` instead of the `#define` precompiler directive
- using `typedef` to create aliases for existing types
- using enumerations combined with `typedef`
- using bit operations to manipulate individual data bits

## 12.1 The comma operator

Expressions can be separated by a comma operator. The expressions are evaluated from left to right and the last one determines the final value.

```
for(i = 0, j = n-1; i < n ; i++, j--)
       b[j] = a[i];
```

After every loop execution the variable i will be increased with 1 and the variable j will be decreased with 1. This example can be rewritten as:

```
for (i = 0; i < n; i++)
       b[n-i-1] = a[i];
```

In the next example, the `scanf` function first reads a new value of n before the condition (n > 0) is checked.

```
sum = 0;
while(scanf("%d%*c", &n), n > 0)
       sum += n;
printf("The sum is %d\n", sum);
```

## 12.2 `typedef`

In C it is possible to create an alias for a previously defined type. To avoid using complex type names, which is the case for enumeration types (see section 12.4) , you can define your own self explaining aliases using the keyword `typedef`.

```
typedef int INTEGER;
```

defines `INTEGER` as an alias for the type `int`. The new type `INTEGER` can be used instead of the standard type `int`:

```
INTEGER counter, size;
```

The above declaration reserves memory for the 2 variables `counter` and `size` that are of the type `INTEGER` or `int`.

## 12.3 Type qualifiers

Different type qualifiers exist in C. In section 6.3, we discussed already the type qualifiers `register` and `static`. Another very useful type qualifier in the C language is `const`.

Constants can be defined in 2 different ways:

- Using the preprocessor directive `#define`:
  this is what we have done up till now. The preprocessor will replace every instance of the constant by the value linked to it in the `#define` directive.
- Using the type qualifier `const`:
  Using the `const` prefix allows to declare constants with a specific type as follows:

  ```
  const type variable = value;
  ```

  Example:
  ```
  const int linelength = 80;
  ```

  this statement assigns the value of 80 to the variable `linelength`. Since the `const` prefix is used, `linelength` can no longer be changed after initialization.

## 12.4 The enumeration type

The `enum` keyword allows to create a new type that consists of a limited list of elements. Variables of such an enumeration type store one of the values of the enumeration set.

For example, the enumeration:

```
enum days {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
```

creates a new type called `enum days`. Variables of that type can hold any one of the identifiers Sun till Sat. Internally, these identifiers are represented by integers starting from 0 for Sun until 6 for Sat.

The declaration:

```
enum days d1, d2;
```

creates 2 variables of the type `enum days`.

Using the keyword `typedef` allows to create an alias for the type `enum days` as follows:

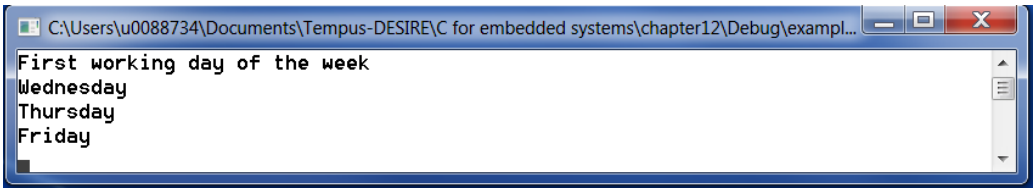
```
typedef enum days {Sun, Mon, Tue, Wed, Thu, Fri, Sat} Day;
```

The declaration of the variables d1 and d2 can now be done by:

```
Day d1, d2;
```

Example:
Print the days of the week using an enumeration type.

```
1   #include <stdio.h>
2
3   typedef enum days { Sun, Mon, Tue, Wed, Thu, Fri, Sat } Day;
4
5   void PrintDay(Day);
6
7   int main(void)
8   {
9      Day d1, d2;
10     d1 = Mon;
11
12     if (d1 == Mon)
13         printf("First working day of the week\n");
14
15     for (d2 = Wed; d2 < Sat; d2++)
16         PrintDay(d2);
17     return 0;
18  }
19
20  void PrintDay(Day d)
21  {
22     switch (d)
23     {
24     case Mon: printf("Monday\n"); break;
25     case Tue: printf("Tuesday\n"); break;
26     case Wed: printf("Wednesday\n"); break;
27     case Thu: printf("Thursday\n"); break;
28     case Fri: printf("Friday\n"); break;
29     case Sat: printf("Saturday\n"); break;
30     case Sun: printf("Sunday\n"); break;
31     }
32  }
```




**Code 58: example enumeration type**

Note that a `switch` is used to translate the identifiers Sun till Sat into real weekdays. Remember that a `switch` statement can only be used if the expression evaluates to an integer number. This is the case since the identifiers are internally stored as integer numbers.

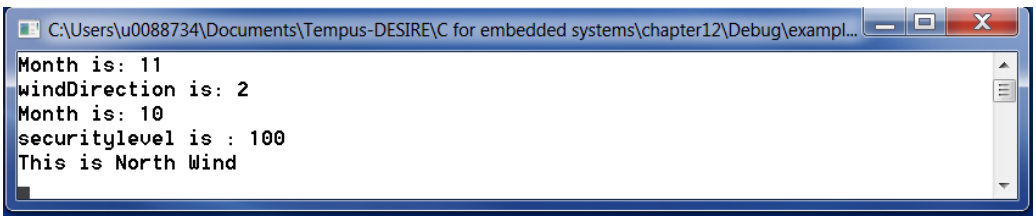
Replacing the for loop in the above example by:

```
for(d2 = Wed; d2 < Sat; d2++)
      printf("%d ", d2);
```

will change the output into: 3 4 5

The integer numbers linked to the different identifiers can also be set explicitly as shown in the example of Code 59:

```
1   #include <stdio.h>
2
3   typedef enum Months_t { JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG,
    SEP, OCT, NOV, DEC } Months_t;
4
5   int main()
6   {
7      enum WindDirections_t { NO_WIND, SOUTH_WIND, NORTH_WIND,
                               EAST_WIND, WEST_WIND };
8      enum WindDirections_t windDirection = NORTH_WIND;
9
10     Months_t month = DEC;
11
12     printf("Month is: %d\n", month);
13     printf("Wind direction is: %d\n", windDirection);
14
15     month = (Months_t)(month - 1);
16     printf("Month is: %d\n", month);
17
18     enum Security_t{TOP_SECURE=100,BLACK_OPS=1000,NO_SECURE=0 };
19     enum Security_t securitylevel = TOP_SECURE;
20     printf("security level is : %d\n", securitylevel);
21
22     switch (windDirection)
23     {
24     case NORTH_WIND:
25           printf("This is North Wind\n");
26           break;
27     case NO_WIND:
28           printf("There is No Wind\n");
29           break;
30     default:
31           printf("Default case\n");
32           break;
33     }
34
35     return 0;
36  }
```




**Code 59: example of enumeration type 2**

The statement:

```
enum Security_t{TOP_SECURE=100, BLACK_OPS=1000, NO_SECURE=0 };
```

defines a new type called `enum Security_t` with identifiers `TOP_SECURE`, stored internally as 100, `BLACK_OPS`, stored internally as 1000 and `NO_SECURE` stored internally as 0.

## 12.5 Bit operations

All data is internally stored as a sequence of bits. Depending on the data type chosen, the number of bits needed to represent the data is different. A character for instance is represented by 1 byte or 8 bits.

Bitwise operators allow to directly manipulate the bits themselves. This can be very useful for instance to manipulate the content of registers inside an embedded system.

An overview of the bitwise operators is shown in Table 9:

| operator | meaning |
|---|---|
| & | bitwise AND |
| \| | bitwise OR |
| << | left shift |
| >> | right shift |
| ~ | one's complement |
| ^ | bitwise XOR |

**Table 9: bitwise operators**

### 12.5.1 Bitwise AND

```
short int w1 = 25, w2 = 77, w3;
w3 = w1 & w2;
```

The bits in `w3` are set to 1 if the corresponding bits in `w1` and `w2` are both 1:

```
w1          0000 0000 0001 1001          25
w2          0000 0000 0100 1101          77
-----------------------------------------
w3          0000 0000 0000 1001           9
```

### 12.5.2 Bitwise OR

```
short int w1 = 25, w2 = 77, w3;
w3 = w1 | w2;
```

The bits in `w3` are set to 1 if at least 1 of the corresponding bits in `w1` and `w2` is 1:

```
w1          0000 0000 0001 1001          25
w2          0000 0000 0100 1101          77
-----------------------------------------
w3          0000 0000 0101 1101          93
```

### 12.5.3 Bitwise XOR

```
short int w1 = 25, w2 = 77, w3;
w3 = w1 ^ w2;
```

The bits in `w3` are set to 1 if the corresponding bits in `w1` and `w2` consist of an odd number of 1's:

```
w1          0000 0000 0001 1001          25
w2          0000 0000 0100 1101          77
-----------------------------------------
w3          0000 0000 0101 0100          84
```

### 12.5.4 One's complement

```
short int w1 = 25, w3;
w3 = ~w1;
```

The bits in `w3` are set to 1 if the corresponding bits in `w1` are set to 0 and vice versa:

```
w1          0000 0000 0001 1001              25
-----------------------------------------------
w3          1111 1111 1110 0110          65 510
```

### 12.5.5 Left shift

```
short int w1 = 25, w3;
w3 = w1 << 2;
```

The bits of `w1` are shifted 2 positions to the left. The last 2 positions on the right are filled with 0:

```
w1          0000 0000 0001 1001           25
-----------------------------------------
w3          0000 0000 0110 0100          100
```

Remark that shifting 2 positions to the left is equivalent to multiplying with $2^2$. In general, shifting n positions to the left is equivalent to multiplying with $2^n$.

### 12.5.6 Right shift

```
short int w1 = 25, w3;
w3 = w1 >> 2;
```

The bits of `w1` are shifted 2 positions to the right. The 2 leftmost positions are filled with 0 or 1 depending on the value of the MSB of `w1`:

```
w1          0000 0000 0001 1001            25
-------------------------------------------
w3          0000 0000 0000 0110             6
```

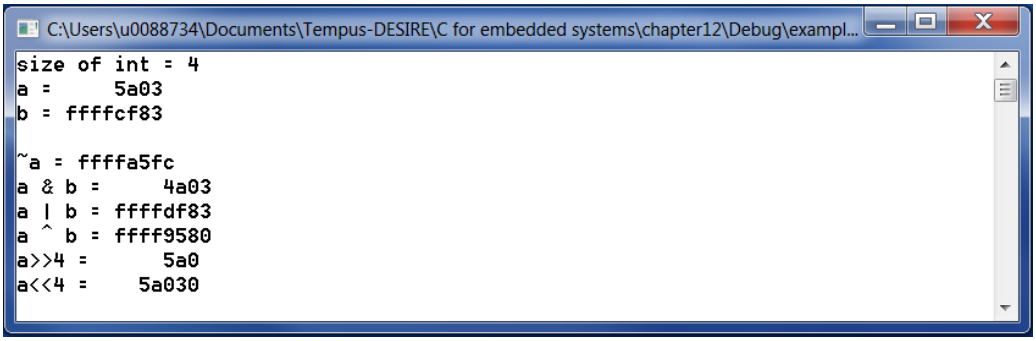
```
short int w2 = 0xCF83, w4;
w4 = w2 >> 2;
```

```
w2          1100 1111 1000 0011          -12 413
-----------------------------------------------
w3          1111 0011 1110 0000            -3 104
```

Remark that shifting 2 positions to the right is equivalent to dividing by $2^2$. In general, shifting n positions to the right is equivalent to dividing by $2^n$.

### 12.5.7 Example

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5      short int a, b;
6
7      a = 0x5A03; // 0101 1010 0000 0011 = 23043
8      b = 0xCF83; // 1100 1111 1000 0011
9
10     printf("size of int = %d\n", sizeof(int));
11     printf("a = %8x\n", a);
12     printf("b = %8x\n\n", b);
13     printf("~a = %8x\n", ~a);            //1010 0101 1111 1100
14     printf("a & b = %8x\n", a & b);      //0100 1010 0000 0011
15     printf("a | b = %8x\n", a | b);      //1101 1111 1000 0011
16     printf("a ^ b = %8x\n", a ^ b);      //1001 0101 1000 0000
17     printf("a>>4 = %8x\n", a >> 4);      //0000 0101 1010 0000
18     printf("a<<4 = %8x\n\n", a << 4);    //0101 1010 0000 0011 0000
19     return 0;
20  }
```




**Code 60: example bit operations**

### 12.5.8 Masking

Consider the example of a 16 bit register out of which only the 4 MSB's (the 4 left most bits) need to be read. Using the above bit operators, one could shift all bits 12 positions to the right such that the 4 MSB's become the only bits left.

Example:
Suppose the register `a` contains the value `23043` (`0101 1010 0000 0011`).
`a>>12` then results in: `0000 0000 0000 0101`
Since leading zero's are not be taken into account, this is equivalent to `101` which is the value of the 4 MSB's.

Unfortunately, this is true only if the MSB of the register `a` is a 0! In that case the leftmost positions will be filled with 0 resulting in leading zero's that can be ignored. However, if the MSB is a 1, shifting the bits 12 positions to the right will result in a series of 12 1's before the bits of interest:

`a = -24523 (1010 0000 0011 0101)`
`a>>12` results in: `1111 1111 1111 1010`, which is equivalent to -6. The 4 MSB's on the other hand are 1010 which represents the number 10!

To get rid of the leading 1's, the result of the shift operation will be treated with a bitmask that turns all bits to zero except for the ones we are interested in. This process is called bit masking.

In the above example the masking can be done by using a bitwise AND operation of the shifted number and the bit sequence that contains 0 on all positions except on the positions corresponding to the bits of interest: `0000 0000 0000 1111` or `0x000F` in hex notation.

`(a>>12) & 0x000F results in 0000 0000 0000 1010`

A programming example that returns the 4 MSB's of a number is shown in Code 61

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5      short number;
6      printf("Enter an integer [-32768,32767]: ");
7      scanf("%hd%*c", &number);
8      printf("Number:\ndec: %hd \nhex: %x.\n\n", number, number);
9      number = number >> 12;     //shift 12 positions to the right
10     printf("number shifted:\ndec: %hd or hex: %x.\n\n", number,
               number);
11     number = number & 0x000f; //masking to remove leading 1's
12     printf("4 MSB's:\ndec: %hd of hex: %x.\n\n", number, number);
13     return 0;
14  }
```

**Code 61: bit masking**

## 12.6 Exercises

**12.1.** Write a program with following screen output:

```
Current Month     | Previous Month
------------------------------
January           | December
Februari          | January
March             | February
April             | March
May               | April
June              | May
July              | June
August            | July
September         | August
October           | September
November          | October
December          | November
```

Make use of:

- an enum type "`month`":
  `enum t_month{Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec}`
- a function "`lastMonth`" that has the current month (type `enum t_month`) as input and returns the previous month
- a function "`PrintMonth`" that takes a variable of the type `enum t_month` as input and prints that month to the screen
- a for loop to run through all 12 months

**12.2.** Write a program "next-day" with:

- a function "`read`" that asks the user to enter a day and a month as integers and returns those values to the main function (hint: use pointers)
- a function "`NextDay`" that has the current day and month as arguments, calculates the next day and returns that next day to the main function (assume the day is not part of leap year)
- a function "`PrintDay`" that takes a day and a month as input and prints them to the screen
- use an `enum` type for the months

```
Enter the current day and month (as integers): 29 3
The current day is: March 29
The next day is: March 30
```

```
Enter the current day and month (as integers): 30 4
The current day is: April 30
The next day is: May 1
```

**12.3.** Write a program that reads an integer, stores it in a variable of the type `short int` and prints the integer as a sequence of 4 nibbles (nibble = 4 bits). The nibbles can be printed in decimal or in hex format. Make sure the leftmost nibble is printed first and the rightmost is printed last. Determining the values of the nibbles needs to be done using bit operations only. Make use of functions in your program.

```
input:      23043 (=0x5A03)
output:     5 10 0 3

input:      -21345 (=0xAC9F)
output:     10 12 9 15
```

**12.4.** Write a function with return value that rotates the bits of a `short int` 4 positions to the left. The bits that are shifted out at the left side, must be reentered at the right side. Rotating needs to be done inside the variable that was entered.
Write a program that uses this function.

```
input:      20480
output:     5
      (20480 = 0x5000 -> rotation yields: 0x0005)

input:      23043
output:     -24523
      (23043 = 0x5A03 -> rotation yields: 0xA035)

input:      -24523
output:     858
      (-24523 = 0xA035 -> rotation yields: 0x035A)
```

**12.5.** Write a function that prints the binary representation of a `short int` using bit operations. Write a program that uses this function.

```
input:      23043
output:     0101 1010 0000 0011

input:      -24523
output:     1010 0000 0011 0101
```

**12.6.** Consider a register of the CANSPI MCP2515 chip. The register contains 8 bits out of which the 2 MSB's (bits 6 and 7) are the "synchronization Jump Width Length bits" and the 6 LSB's (bits 0 till 5) are the "Baud Rate Prescaler bits" (see figure below)

**REGISTER 5-1: CNF1 – CONFIGURATION 1 (ADDRESS: 2Ah)**

| R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 |
|---|---|---|---|---|---|---|---|
| SJW1 | SJW0 | BRP5 | BRP4 | BRP3 | BRP2 | BRP1 | BRP0 |
| bit 7 | | | | | | | bit 0 |

| **Legend:** | | | |
|---|---|---|---|
| R = Readable bit | W = Writable bit | U = Unimplemented bit, read as '0' | |
| -n = Value at POR | '1' = Bit is set | '0' = Bit is cleared | x = Bit is unknown |

bit 7-6 **SJW<1:0>:** Synchronization Jump Width Length bits
11 = Length = 4 x TQ
10 = Length = 3 x TQ
01 = Length = 2 x TQ
00 = Length = 1 x TQ

bit 5-0 **BRP<5:0>:** Baud Rate Prescaler bits
TQ = 2 x (BRP + 1)/Fosc

Write a program that asks the user to enter a number that fits this register (1 byte) and that prints the corresponding "Baud Rate Prescaler bits" (hex and binary format) and the "Synchronization Jump Width bits" (hex and binary format).

**12.7.** Write a function that produces a (pseudo) random number in the interval [1, 32767]. Make use of the following algorithm:

- use a SEED different form 0 (ex: 3254)
- store this seed number into a 16 bit `static short int` variable (static is needed to continue with the changed value when the function is called several times)
- perform a XOR operation on the bits 14 and 13 (the bits are numbered from left to right as follows: 15, 14, 13, …, 2, 1, 0)
- shift the number 1 bit to the left and fill the rightmost bit with the result of the XOR operation
- make sure bit 15 remains 0 (to keep the number in the correct range)
- the number you have now is a pseudo random number

Write a program that calls this function 10 times. Check the result.

# 13 The C preprocessor

**Objectives**

This chapter explains how preprocessor directives can be used to:
- include files like a self-written header file
- define symbolic constants
- write macros
- use conditional compilation to specify which portions of the code need to be compiled in specific situations only

## 13.1 The C preprocessor

As explained in section 1.5, a program written in C code needs to traverse different steps before it can be executed. First, the C preprocessor or parser will search for preprocessor directives in the source code and take the appropriate action. Next, the compiler will translate the code produced by the preprocessor into an object-file. Finally, the linker will create an executable.

Preprocessor directives always begin with `#`. Only white spaces or comments can be written on the same line before a preprocessor directive.

## 13.2 `#define` preprocessor directive

`#define` can be used to define both symbolic constants and macros.

### 13.2.1 Symbolic constants

The `#define` preprocessor directive allows to allocate symbolic names to constants used inside the program.

The general format is:

```
#define symbolic-name replacement-text
```

The result of this directive is that the preprocessor will look for all occurrences of the `symbolic-name` in the source code and replace each occurrence with the `replacement-text` before the source code is compiled.

```
#define MAX 100
#define YES 1

if(i < MAX)
       answer = YES;
```

In the above example, the word `MAX` will be replaced with the value 100 and the word `YES` with the value 1.

The preprocessor directive `#undef` discards the symbolic constant from there on.

```
#undef MAX
```

**Common mistake**

Preprocessor directives are different from C statements. Therefore, no semicolon can be placed at the end of such a directive.

**Common mistake**

No assignment operator can be placed in between the symbolic name and the corresponding replacement text! Doing so results in strange behavior! For instance:

```
#define MAX = 100
```

will cause the preprocessor to replace every occurrence of the word MAX with the replacement text "= 100"!

### 13.2.2 Macros

A macro is basically a symbolic constant with arguments, defined in a `#define` preprocessor directive.

Like for a symbolic constant, all occurrences of the macro name will be replaced by the replacement text. The only difference is the presence of the arguments that will be substituted in the replacement text before replacing the macro name occurrences.

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
```

Wherever `MAX(x,y)` appears in the file, the values of x and y will replace the letters a and b in the macro. Afterwards, the full replacement text with a and b substituted, will replace `MAX(x,y)`.

For instance the statement:

```
y= 2 * MAX(i+1, j-1);
```

will be replaced by:

```
y = 2 * ((i+1) > (j-1) ? (i+1) : (j-1));
```

The parenthesis around every a and b in the replacement text are needed to force the correct order of evaluation when the macro arguments happen to be an expression rather than a single value.

Consider for example the following macro definition:

```
#define SQUARE(x) x * x
```

The statement

```
x = SQUARE(a+b);
```

will be replaced by

```
x = a+b * a+b;
```

Taking the rules of precedence into account, the above expression is equivalent to:

```
x = a + (b * a) + b;
```

which is not the same as $(a+b)^2$ ! Using parenthesis around every x results in the macro definition:

```
#define SQUARE(x) (x)*(x)
```

In this case the above statement will be replaced by:

```
x = (a+b)*(a+b);
```

as intended.

**Common mistake**

No white space can be written in between the macro name and its list of arguments.

**Common mistake**

Putting too little parenthesis around the macro arguments in the replacement text is a commonly made error that can result in strange program output.

Be careful in using macro's that evaluate their arguments more than once like in the macro `SQUARE` defined above. If the macro argument is an expression that changes the variable value like for instance incrementing the variable, the change will be carried out more than once! For instance, the statement

```
x = SQUARE(++a);
```

will be replaced by

```
x = (++a)*(++a);
```

Using functions is safer in this case.

Macro examples:

```
#define IS_LEAPYEAR(j) ((j)%4 == 0 && (j)%100 != 0 || (j)%400 == 0)

#define SWAPINT(x,y) (int help=x; x=y; y=help;)

#define ISDIGIT(c) ((c)>='0' && (c)<='9')

#define TOLOWER(c) ((c) - 'A' + 'a')
```

**Remark**

A macro is different from a function! Using a macro will result in a text substitution, not in a function call!

## 13.3 `#include` preprocessor directive

With the `#include` preprocessor directive a copy of the specified file will be included in the source code.

The 2 standard formats for the `#include` directive are:

```
#include <filename>
#include "filename"
```

In the first case, the filename is enclosed in angle brackets (< >). This instructs the preprocessor to search for the file with name `filename` in the standard include directory. If the file is not found, a preprocessor error will be issued.

If the filename is enclosed in double quotes (" "), like in the second format, the preprocessor will look for the file in the directory that contains the source code file. If not found in that directory, the standard include directory will be searched instead. This is typically used to include programmer defined header files:

```
#include "header.h"
```

Programmer defined header files contain preprocessor definitions, declarations of structures, function prototypes, enumerations, typedef's and, if needed, global variables.

## 13.4 Conditional compilation

Conditional compilation allows to control which preprocessor directives are carried out and what portion of the source code will be compiled. A conditional preprocessor directive can be followed by a constant integer expression only.

### 13.4.1 `#ifdef` preprocessor directive

The general format of the `#ifdef` preprocessor directive is:

```
#ifdef <identifier>
       C code to be compiled if the identifier has been defined
#else
       C code to be compiled if the identifier has not been defined
#endif
```

This is often used to enclose `printf` statements that are only to be compiled if a debug constant is set:

```
#define DEBUG

int main(void)
{
    …
    #ifdef DEBUG
       printf("the values of the variables at this point are: … ", …);
    #endif
    …
    return 0;
}
```

If the symbolic constant `DEBUG` is not set, the `printf` statement will not be compiled and as such it will not be part of the executable.

An example of using `#ifdef` to control the execution of preprocessor directives is shown below:

```
#ifndef PI
#define PI  3.14159265358979
#endif
```

### 13.4.2 `#if` preprocessor directive

The usage of `#if` is similar to the one of `#ifdef` except that `#if` can be followed by a constant expression:

```
#define TEST 1

int main(void)
{
    …
#if TEST
       printf(…);
#endif
    return 0;
}
```

## 13.5 Exercises

**13.1.** Define a macro with name `MIN` to determine the smallest of 2 values. Write a program that tests the macro.

Example:

```
The minimum of -9 and 10 = -9
```

**13.2.** Define a macro `TOLOWER(c)` that changes c into a small letter if c is a capital letter or leaves c as is otherwise. Write a program that tests the macro.

Example:

```
Enter characters, end with 0: A B D c d 0

a b d c d
```

**13.3.** Define a macro `MAX3` that determines the maximum of 3 values. Write a program that tests the macro.

Hint: define a macro `MAX2` that determines the maximum of 2 values and use it as follows:
`MAX3(a, b, c) = MAX2(MAX2(a, b), c)`

Example:

```
The maximum of 20, 10 and -5 = 20
```

**13.4.** Define macro's `IS_CAPITAL` and `IS_SMALL` that result in a 0 if the character entered is a capital or a small letter respectively. Write a program that tests the macro's.

Example:

```
Is A a capital letter? 1
Is A a small letter? 0
```

**13.5.** Define a macro `IS_LETTER` that results in the value 1 if the argument is a letter. Use the macro's from exercise 13.4 to define this macro. Write a program that tests the macro's.

Example:

```
Is a a letter? 1
Is * a letter? 0
```

# 14 File handling in C

**Objectives**

This chapter explains how files can be accessed from a C program. At the end of this chapter, one should be able to:
- understand the difference between text files and binary files
- open and close a file in different modes
- read data from a text file and a binary file
- write data to a text file and a binary file
- update data in binary files

## 14.1 File pointer

All data used in the programs written up till now, was stored in variables. Unfortunately, this type of storage is temporary and as such all data is lost as soon as the program terminates. To allow for long term storage, files need to be used.

A file can be described as a sequence of bytes that is stored on a secondary storage device, which is generally some kind of disk. Hence, to manipulate files, the program needs to store all kinds of information about those files. To this end, a special structure called `FILE` is defined in `<stdio.h>`. When opening a file, a new file control block of the type `FILE` will be created and a pointer to that block (`FILE *`) will be provided.

C provides no specific I/O statements, therefore a series of functions was added to the standard library `stdio`. Some of those functions will be handled in more detail in the next sections.

Every file manipulation in C consists of the following 3 steps:

1. Open the file
2. Read and or write operations
3. Close the file

We will first discuss how to open and close a file. Afterwards different functions to read from files or write data into a file will be explained.

## 14.2 Opening and closing a text file

Creating a new file control block of the type `FILE` and initializing all fields of that block with the correct data is done using the function `fopen`:

```
FILE * = fopen(char * name, char * mode);
```

The function `fopen` has 2 input parameters.

The first parameter is a string that defines the name of the file to be opened. This filename can include path information in an absolute or relative way.

**Common mistake**

If path information is to be added to the file name, be aware that the backslash sign "`\`" has a special meaning when used in a literal string. Therefore, every directory separation character "`\`" needs to be written twice!

The second parameter "`mode`" defines how the file needs to be opened. The mode parameter needs to be written with a small letter, in between double quotes and must be one of the options listed in Table 10.

| mode | description |
|---|---|
| "`r`" | Open an **existing** file for **reading**. |
| "`w`" | Create a **new file** for **writing**. If the file already exists, erase the current content. |
| "`a`" | **Append** data at the end of the file. If the file does not exist, it is first created. |
| "`r+`" | Open an **existing** file for update (**reading** and **writing**). |
| "`w+`" | Create a **new file** for **reading** and **writing**. If the file already exists, erase the current content. |
| "`a+`" | Open an existing file or create a new file for **reading** and **appending**. Writing operations are carried out at the end of the file, for reading operations, repositioning is possible. |

**Table 10: file opening modes**

If the file was opened successfully, the function `fopen` returns a pointer of the type `FILE *` to the newly created file control block. If not, the value `NULL` is returned instead.

Once all read and write operations on the file are done, the file control block can be cleared by closing the file with the function `fclose`:

```
int fclose(FILE * filepointer);
```

where `filepointer` points to the file control block you want to erase. If the file closure was successful, the function `fclose` returns 0. If not, the value `EOF` is returned.

The usage of the functions `fopen` and `fclose` is demonstrated in Code 62.

```
1   #include <stdio.h>     // contains I/O functions
2   #include <stdlib.h>    // contains function exit()
3
4   int main(void)
5   {
6      FILE * fp;
7      fp = fopen("MyFile.txt", "r");
8      if (fp == NULL)
9      {
10            printf("Error opening file \"MyFile.txt\" for read.\n");
11            if ((fp = fopen("MyFile.txt", "w")) == NULL)
12            {
13                  printf("Error creating file \"MyFile.txt\".\n");
14                  exit(1);
15            }
16            printf("New file \"MyFile.txt\" was created.\n");
17     }
18     else
19     {
20            printf("\"MyFile.txt\" was successfully opened\n");
21     }
22     fclose(fp);
23     return 0;
24  }
```

**Code 62: opening and closing a file**

With the statement

```
fp = fopen("MyFile.txt", "r");
```

a new file control block is made, the file pointer `fp` now points to that file control block. To avoid malfunctioning of the program in case for instance the file does not exist yet, the program checks if `fp` has valid content before starting any file operation:

```
if( fp == NULL ){ … }
```

Both operations can be combined into 1 single instruction:

```
if((fp=fopen("MyFile.txt", "r")) == NULL)
{
      printf("Error opening file \"MyFile.txt\" for read.\n");
      exit(1);
}
```

In case of an error during the opening operation, the function `exit()` will simply stop the program in a controlled way. It is good practice to print a line of text explaining why the program stopped before actually exiting the program.

**Common mistake**

Omitting the parenthesis around the expression

`fp=fopen("MyFile.txt", "r")`

is a commonly made mistake. As a result, the filepointer `fp` will not be linked correctly to the opened file.

## 14.3 Read and write 1 symbol to a text file

### 14.3.1 Read one symbol: `fgetc`

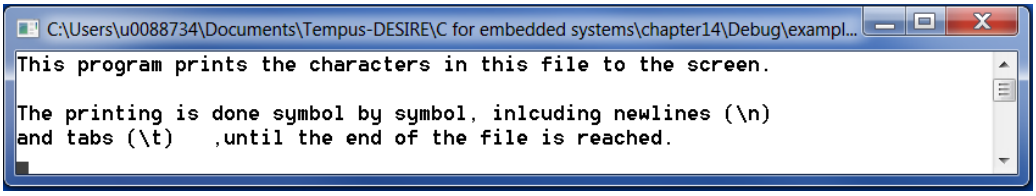
The function `fgetc`, with function declaration:

```
int fgetc(FILE *fp);
```

reads one character from the file referenced by `fp` and returns the ASCII value of that character as an integer. If the end of the file is reached or if a reading error occurred, the function `fgetc` returns the constant `EOF`. To check which one of the 2 occurred, the functions `ferror` and `feof` can be used.

The following example shows how the text of a file can be printed to the screen using the function `fgetc`.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   #define FILENAME "text.txt"
5
6   int main(void)
7   {
8      FILE * fp;
9      int symbol;
10
11     if ((fp = fopen(FILENAME, "r")) == NULL)
12     {
13            printf("The file \"%s\" cannot be opened.\n", FILENAME);
14            exit(1);
15     }
16
17     while ((symbol = fgetc(fp)) != EOF)
18            putchar(symbol);
19
20     printf("\n");
21     fclose(fp);
22     return 0;
23  }
```




**Code 63: read 1 symbol from a file**

### 14.3.2 Write one symbol: `fputc`

The function `fputc`, with function declaration:

```
int fputc(int char, FILE *fp);
```

writes one character to the file referenced by `fp`. The function `fputc` takes the ASCII value of the character to be written (`int char`) as input and returns that ASCII value, except if a writing error occurred. In the case of an error, the value `EOF` will be returned.

Changing the program of Code 63 to make a file copy results in:

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   #define IN "text.txt"
5   #define OUT "copy.txt"
6
7   int main(void)
8   {
9      FILE *fin, *fout;
10     int symbol;
11
12     if ((fin = fopen(IN, "r")) == NULL)
13     {
14            printf("The file \"%s\" cannot be opened.\n", IN);
15            exit(1);
16     }
17
18     if ((fout = fopen(OUT, "w")) == NULL)
19     {
20            printf("The file \"%s\" cannot be opened.\n", OUT);
21            exit(2);
22     }
23
24     while ((symbol = fgetc(fin)) != EOF)
25            fputc(symbol, fout);
26
27     fclose(fin);
28     fclose(fout);
29     printf("The file was copied.\n");
30     return 0;
31  }
```

**Code 64: copy a file using fgetc and fputc**

First, the two files are opened in read and write mode respectively. In every execution of the `while` loop one symbol of the file "`text.txt`" is read (`fgetc(fin)`) and written into the file "`copy.txt`" (`fputc(symbol, fout)`).

## 14.4 Read and write a full line to a text file

### 14.4.1 Read a full line: `fgets`

The function `fgets`, with function declaration:

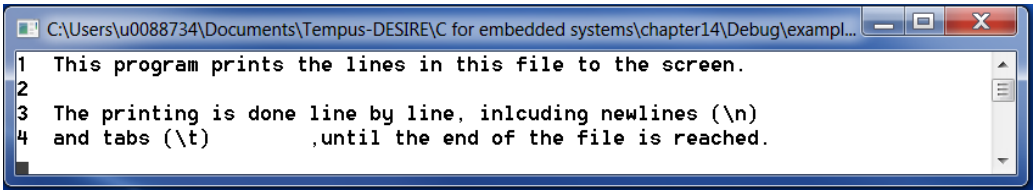
```
char * fgets(char *s, int max, FILE *fp);
```

reads a sequence of symbols from the file referenced by `fp`, and stores them in the string `s`. The read process stops when a newline character is found, the maximal number of symbols defined by the parameter `max` is read (null byte included) or the end of the file is reached. At the end of the symbol sequence a null byte is added automatically.

The function returns the string that was read except if the end of the file is reached or if a reading error occurred. In that case, the function `fgets` returns the constant `NULL`. To check which one of the 2 occurred, the functions `ferror` and `feof` can be used.

The following example reads a text file line by line and prints the lines to the screen.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   #define FILENAME "text.txt"
5
6   int main(void)
7   {
8      FILE * fp;
9      int i = 0;
10     char line[BUFSIZ];  // BUFSIZ is defined in stdio.h to be 512
11
12     if ((fp = fopen(FILENAME, "r")) == NULL)
13     {
14            printf("The file \"%s\" cannot be opened\n", FILENAME);
15            exit(1);
16     }
17
18     while (++i, fgets(line, BUFSIZ, fp) != NULL)
19            printf("%d  %s", i, line);
20
21     fclose(fp);
22     printf("\n");
23     return 0;
24  }
```




**Code 65: printing the content of a file line by line**

### 14.4.2 Write a full line: `fputs`

The function `fputs`, with function declaration:

```
int fputs(char *s, FILE *fp);
```

writes a sequence of symbols, stored in the string `s`, to the file referenced by `fp`. If successful, the function returns a non-negative value, otherwise `EOF` is returned.

The example of Code 66 reads a line of text from the keyboard and appends it to a file.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   #define FILENAME "text.txt"
5
6   int main()
7   {
8      FILE * pFile;
9      char sentence[BUFSIZ];
10
11     printf("Enter sentence to append: ");
12     gets(sentence);
13
14     if ((pFile = fopen(FILENAME, "a")) == NULL)
15     {
16            printf("The file \"%s\" cannot be opened\n", FILENAME);
17            exit(1);
18     }
19
20     fputs(sentence, pFile);
21     fputc('\n', pFile);        //add a newline after the sentence
                                    into the file
22     fclose(pFile);
23     return 0;
24  }
```

**Code 66: append a line of text to an existing text file**

Note that a newline character is written explicitly to the file to avoid writing all sentences on 1 single line. In other words: `fputs` does not automatically put a newline character at the end of the string written.

## 14.5 Formatted read and write to a text file

Formatted writing to and reading from files is done with the functions `fprintf` and `fscanf` respectively that are very similar to the functions `printf` and `scanf`.

### 14.5.1 Formatted printing to a file: `fprintf`

The function `fprintf` is declared in `<stdio.h>` as:

```
int fprintf(FILE *fp, format control string, arguments);
```

except for the first argument that indicates the file to write to, this is identical to the `printf` function. Beside strings, using `fprintf` also allows other data types to be written to the file. This is shown in example Code 67.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   #define FILENAME "text.txt"
5
6   int main(void)
7   {
8      FILE * fp;
9
10     char name1[12] = "Smith";
11     int age1 = 55;
12     float weight1 = 99.56;
13     char name2[12] = "Jones";
14     int age2 = 45;
15     float weight2 = 56.78;
16
17     if((fp = fopen(FILENAME, "w"))==NULL)
18     {
19            printf("The file \"%s\" cannot be opened\n", FILENAME);
20            exit(1);
21     }
22
23     fprintf(fp, "%s %d %f\n", name1, age1, weight1);
24     fprintf(fp, "%s\t%d\t%f\n", name2, age2, weight2);
25
26     fclose(fp);
27     printf("The file was written\n");
28     return 0;
29  }
```

**Code 67: formatted printing to a file**

### 14.5.2 Formatted reading from a file: `fscanf`

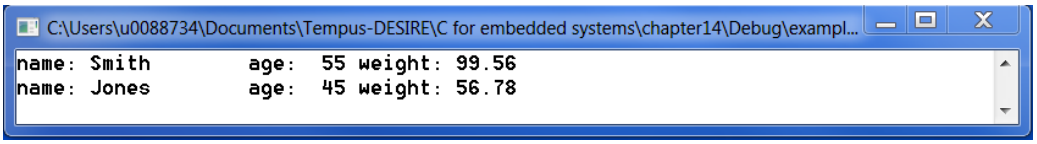
Reading strings and symbols can be done using the functions `fgetc` and `fgets` as described above. However, if only one word needs to be read or if a number needs to be read, formatted reading is needed. To this end, the function `fscanf` is defined as:

```
int fscanf( FILE *fp, format control string, arguments);
```

Also this function is identical to its `scanf` equivalent except for the file pointer. The return value of this function is a non-negative integer if successful.

Code 68 shows the C code needed to read the file written in Code 67 and print it to the screen.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   #define FILENAME "text.txt"
5
6   int main(void)
7   {
8      FILE * fp;
9
10     char name[12];
11     int age;
12     float weight;
13
14     if ((fp = fopen(FILENAME, "r")) == NULL)
15     {
16            printf("The file \"%s\" cannot be opened\n", FILENAME);
17            exit(1);
18     }
19
20     while (fscanf(fp, "%s%d%f", name, &age, &weight) > 0)
21     {
22            printf("name: %-12s age: %3d weight: %5.2f\n", name,
                     age, weight);
23     }
24
25     fclose(fp);
26     printf("\n");
27     return 0;
28  }
```




**Code 68: formatted reading from a file**

The format control string `"%s%d%f"` indicates that first a string, then an integer and finally a float must be read and stored in the addresses given by the arguments `name`, `&age` and `&weight`.

Note that the spaces or tabs in between the values printed in the file are not taken into account by the `fscanf` function.

## 14.6 `stdin`, `stdout` and `stderr`

When program execution begins, 3 files are opened automatically with associated `FILE` pointers `stdin`, `stdout` and `stderr`. `stdin` points to the standard input, `stdout` to the standard output and `stderr` to the standard error.

As such, the statement `printf("Hello world\n");`
is identical to `fprintf(stdout, "Hello world\n");`
and `getchar();` can be replaced by `fgetc(stdin);`

Similarly `scanf()`, `gets()` and `puts()` can be substituted by their corresponding file functions.

The principle of connecting files to the standard input and output also allows to redirect them to a file starting from the command line. If for instance `prog` is an executable, the command line input:

```
prog > out
```

will run the program with name `prog` and redirect the standard output to the file with name `out`. Similarly the command:

```
prog < in
```

makes the program `prog` read all standard input from the file with name `in`.

## 14.7 Binary files versus text files

Text files are used to make the information in the file readable to the user. However, many files in a computer system contain information that is only to be read or updated by the system itself. As a result, this information does not need to be formatted and can be copied straight from memory into the file. This results in files containing a sequence of bytes that are not necessarily to be interpreted as text characters. Opening such a binary file with a text editor is usually not very useful.

## 14.8 Opening and closing a binary file

The functions `fopen` and `fclose` are used to open and close binary files:

```
FILE * = fopen(char * name, char * mode);
int fclose(FILE * filepointer);
```

As with text files, the first parameter of the function `fopen` is a string that defines the name of the file to be opened and can include path information.

The second `fopen` parameter "`mode`" defines how the file needs to be opened. The possible mode options are identical to the ones used for text files except that the letter 'b' is added at the end to indicate that a binary file is being opened. This is illustrated in Table 11.

| mode | description |
|---|---|
| "rb" | Open an **existing** binary file for **reading**. |
| "wb" | Create a **new binary file** for **writing**. If the file already exists, erase the current content. |
| "ab" | **Append** data at the end of the binary file. If the file does not exist, it is created first. |
| "rb+" | Open an **existing** binary file for update (**reading** and **writing**). |
| "wb+" | Create a **new binary file** for **reading** and **writing**. If the file already exists, erase the current content. |
| "ab+" | Open an existing binary file or create a new binary file for **reading** and **appending**. Writing operations happen at the end of the file, for reading operations, repositioning is possible. |

**Table 11: binary file opening modes**

## 14.9 Write to a binary file: `fwrite`

The function `fwrite`, with function declaration:

```
size_t fwrite(void *ptr, size_t size, size_t count, FILE *fp);
```

writes `count` elements of `size` bytes starting from the address `ptr` from memory into the file corresponding to the filepointer `fp`. The return value is an `unsigned int` (`size_t`) that indicates the amount of elements written and is equal to 0 if an error occurred.

The meaning of each argument is listed in Table 12:

| argument | description |
|---|---|
| ptr | starting address in memory |
| size | size (in bytes) of 1 element |
| count | number of elements (`size` bytes each) |
| fp | filepointer |

**Table 12: arguments of function `fwrite`**

Example: write a series of 3 floats to a binary file.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   #define FILENAME "myfile.bin"
5
6   int main(void)
7   {
8      FILE* pFile;
9      float buffer[3] = { 1.0, 11.5, 48.45 };
```

```
10      pFile = fopen(FILENAME, "wb");
11      if (pFile == NULL)
12      {
13              printf("The file %s cannot be opened.\n", FILENAME);
14              exit(1);
15      }
16
17      fwrite(buffer, sizeof(float), 3, pFile);
18      fclose(pFile);
19      return 0;
20  }
```

**Code 69: `fwrite` example**

In the above example, `buffer` is the name of an array so it represents the starting address of that array in memory. To determine the number of bytes needed for 1 element, in this case 1 `float`, the function `sizeof` is used.

Opening "`myfile.bin`" in a text editor results in unreadable content:

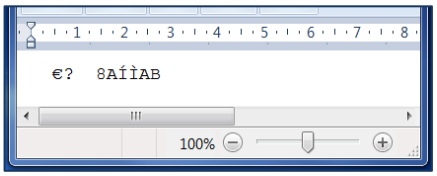



**Figure 48: reading binary file with text editor**

Reading binary files can be done using a hex editor. Such a program shows the bytes stored in the file in hexadecimal mode:

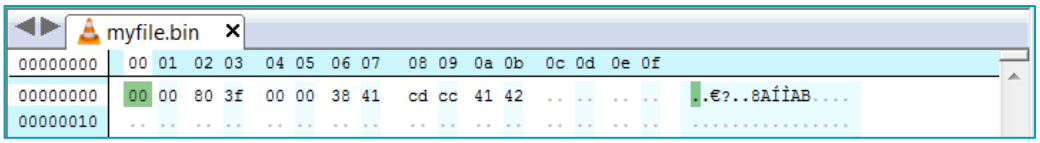



**Figure 49: reading binary file with hex editor**

The numbers in the above figure are the hexadecimal representations of the floats written. Reading the content of such a binary file is easier using the function `fread` and the build-in `printf` formatting to print the numbers in readable format to the screen.

## 14.10 Read from a binary file: `fread`

The function `fread`, with function declaration:

```
size_t fread(void *ptr, size_t size, size_t count, FILE *fp);
```

reads `count` elements of `size` bytes from the file corresponding to the filepointer `fp` to the memory location `ptr`. The return value is an `unsigned int` (`size_t`) that indicates the amount of elements read and is equal to 0 if an error occurred or the end of the file is reached.

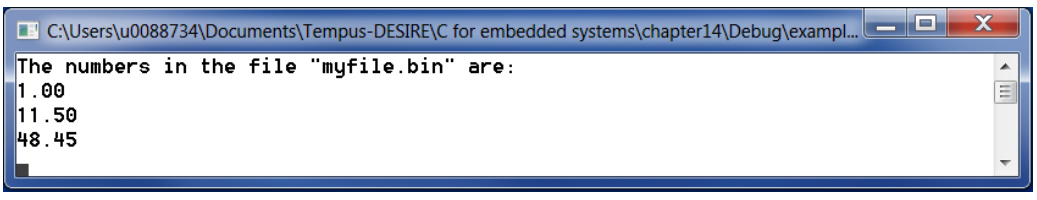
The meaning of each argument is listed in Table 13.

| argument | description |
|---|---|
| ptr | address in memory |
| size | size (in bytes) of 1 element |
| count | number of elements (`size` bytes each) |
| fp | filepointer |

**Table 13: arguments of function `fread`**

Example: read the file "`myfile.bin`" and print its content to the screen.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   #define FILENAME "myfile.bin"
5
6   int main(void)
7   {
8      FILE* pFile;
9      float number;
10
11     if ((pFile = fopen(FILENAME, "rb")) == NULL)
12     {
13            fputs("File error\n", stderr);
14            exit(1);
15     }
16
17     printf("The numbers in the file \"%s\" are:\n", FILENAME);
18     while (fread(&number, sizeof(float), 1, pFile))
19            printf("%.2f\n", number);
20
21     fclose(pFile);
22     return 0;
23  }
```




**Code 70: `fread` example**

## 14.11 More binary file functions

### 14.11.1 function `fseek`

The function `fseek`, with function declaration:

```
int fseek(FILE * fp, long offset, int origin);
```

places the file position pointer for the file referenced by `fp` to the byte location that is the sum of `origin` and `offset`. The return value is equal to 0 unless an error occurred. In the case of an error the value -1 is returned.

The argument `origin` must be one of the values listed in Table 14.

| origin | description |
|---|---|
| SEEK_SET | beginning of the file |
| SEEK_CUR | current location in the file |
| SEEK_END | end of the file |

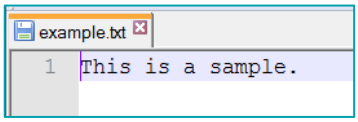
**Table 14: list of values to be used as origin in the function `fseek`**

Code 71 shows an example of `fseek` usage.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #define FILENAME "example.txt"
4
5   int main(void)
6   {
7      FILE* pFile;
8      pFile = fopen(FILENAME, "w");
9      if (pFile == NULL)
10     {
11           printf("The file %s cannot be opened.", FILENAME);
12           exit(1);
13     }
14
15     fputs("This is an apple.", pFile);
16     fseek(pFile, 9, SEEK_SET);
17     fputs(" sam", pFile);
18     fclose(pFile);
19     return 0;
20  }
```

**Code 71: `fseek` example**

The statement "`fseek(pFile, 9, SEEK_SET);`" places the file position pointer at an offset of 9 bytes (and thus 9 letters) from the beginning of the file. The next statement "`fputs(" sam", pFile);`" starts writing the symbols `" sam"` from byte 9 on, overwriting the content that is already there, resulting in following content for the file "example.txt":



### 14.11.2 function `ftell`

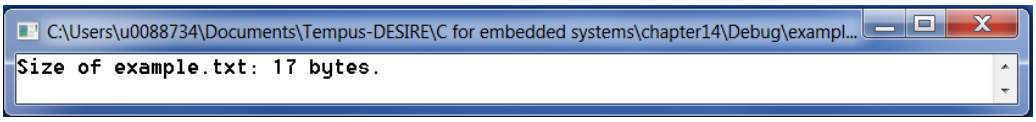
The function `ftell`, with function declaration:

```
long ftell(FILE * fp);
```

returns the current offset of the file position pointer in the file referenced by `fp`, with respect to the beginning of the file. This offset is expressed in bytes. If an error occurs, the return value is `-1L`.

Using `ftell` to determine the length of a file is illustrated in Code 72.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #define FILENAME "example.txt"
4
5   int main(void)
6   {
7      FILE* pFile;
8      long size;
9
10     if ((pFile = fopen(FILENAME, "rb")) == NULL)
11           fputs("Error opening file\n", stderr);
12     else
13     {
14           fseek(pFile, 0, SEEK_END);  //put position pointer at end
                                           of the file
15           size = ftell(pFile);        //ask current position of
                                           position pointer
16           fclose(pFile);
17           printf("Size of %s: %ld bytes.\n", FILENAME, size);
18     }
19     return 0;
20  }
```




**Code 72: `ftell` example**

## 14.12 Direct access files

Binary files are often used to read and write arrays of data to and from disk. More specifically, full structures can be written and read. We will treat the subject of structures in chapter 15.

Such a structure can be used to represent certain records you want to save like for instance all bank information from a certain person, address information form a customer, …

Since full records can be written at once to a binary file and since we know the length of 1 record, the data from 1 specific record can be accessed by jumping directly to the wanted record. Using fseek, we can put the file position pointer at the right place in the file as is illustrated in Figure 50.

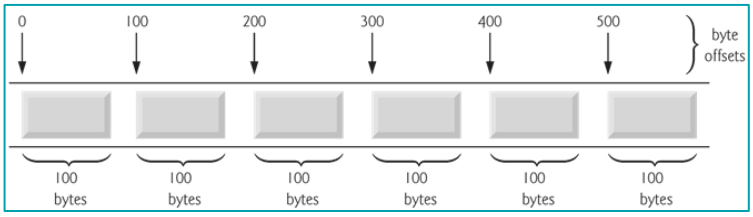



**Figure 50: direct access file**

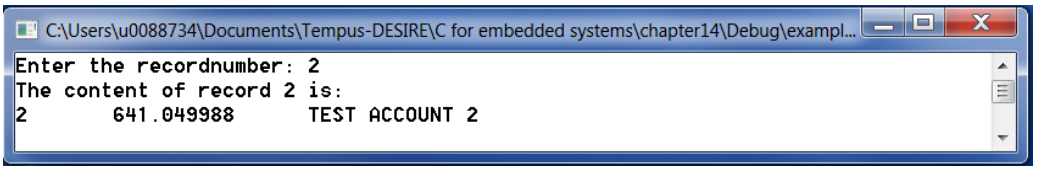
Therefore, binary files are often called direct access files.

Example:
the file records.dat contains the bank balances of 3 customers. Each one of those bank balances is contained in a record with fixed length. The example below, shows how to read the data from 1 record only.

```
1   #include<stdio.h>
2   #include<stdlib.h>
3   #include "account.h"
4
5   int main(void)
6   {
7      FILE * fp;
8      Account account;
9      int recnr;
10
11     if ((fp = fopen(FILENAME, "rb")) == NULL)
12     {
13            printf("The file %s cannot be opened\n", FILENAME);
14            exit(1);
15     }
16
17     printf("Enter the record number: ");
18     scanf("%d%*c", &recnr);
19
20     fseek(fp, (long)(recnr-1) * sizeof(Account), SEEK_SET);
21
22     if (fread(&account, sizeof(Account), 1, fp) <= 0)
23     {
24            printf("Record %d of file %s does not exist.\n", recnr,
                     FILENAME);
25            exit(1);
26     }
27
28     printf("The content of record %d is:\n%d\t%f\t%s\n", recnr,
              account.nr, account.balance, account.name);
29     fclose(fp);
30     return 0;
31  }
```

With account.h:

```
1   #ifndef _ACCOUNT_H
2   #define _ACCOUNT_H
3   #define FILENAME "records.dat"
4   typedef struct         //used to define the fields of 1 record
5   {
6      int nr;
7      char name[25];
8      float balance;
9   } Account;
10  #endif
```




**Code 73: usage of direct access files**

Based upon the record number entered, the file position pointer is set at the start of the wanted record by the statement:

```
fseek(fp, (long)(recnr-1) * sizeof(Account), SEEK_SET);
```

where `sizeof(Account)` evaluates to the length of 1 record. Next, the number of bytes equal to 1 record is read and stored in the variable `account`. Afterwards, the different record fields are printed. A variable like `account` that contains fields of different data types is called a structure. Structures are explained in chapter 15.

## 14.13 Exercises

**14.1.** Use a text editor to create a text file that contains a few lines of text. Write a program that reads the text file line by line and writes everything to the screen.

**14.2.** Write a program that copies the text file line by line to a new file.

**14.3.** Write a program that copies the text file but this time with double spacing. (instead of 1 newline, 2 newlines need to be placed at the end of every line).

**14.4.** Write a program that copies the text file from exercise 14.3 to a new file removing all empty lines.

**14.5.** Write a program that writes the lines of a text file to the screen. On the screen, all lines need to be preceded by their line number.

**14.6.** Write a program that prints the content of 2 different text files merged (one after the other) to the screen.

**14.7.** Write a program that asks the user to enter the file name, the file extension and some text. The program creates that file and writes the entered text to that file.

```
Enter file name: test
Enter file extension: txt
Enter your text: This is a test.

File has been created!
Text has been written
File closed
```

**14.8.** Write a program that prints the first 20 lines of a text file to the screen. If the user hits return on the keyboard, the next 20 lines are printed. This is repeated until the end of the file is reached.

**14.9.** Use a text editor to create 2 different files that both contain a row of integer numbers, ordered from small to large. Write a program that prints the numbers of both files in 1 ordered row. The file content cannot be saved into arrays to solve this problem.

Example:

```
numbers1.txt:        1 5 7 9 11 23 77 93 103
numbers2.txt:        4 9 12 28 124 230
yields: 1 4 5 7 9 9 11 12 23 28 77 93 103 124 230
```

**14.10.** Use a text editor to create 2 different files that both contain a row of integer numbers, ordered from small to large. Write a program that combines the numbers in both files to create 1 ordered row and writes these numbers in that order to a new file. Again, no arrays can be used.

**14.11.** Write a program that searches patterns in a text file. A certain string (pattern) needs to be searched in the text file. Every line that contains that pattern, must be printed. Print also the line number.

**14.12.** Write a program that asks the user to enter the name and age of a chosen number of people and writes that data to a file. The names do not contain any spaces.

```
For how many people do you want to enter data? 3
Enter name and age: Smith 40
Enter name and age: Connelly 25
Enter name and age: Jolie 39
```

results in a file containing:

```
name: Smith
age: 40

name: Connelly
age: 25

name: Jolie
age: 39
```

**14.13.** Write a program that reads the file created in exercise 14.12 and prints all data to the screen.

**14.14.** Write a program that searches a name entered by the user in the file created in exercise 14.12 and prints the corresponding age to the screen. Make sure a message is printed when the name is not present in the file.

```
Enter the name of the person you want to find: Jolie

name: Jolie
age: 39
```

# 15 Structures

**Objectives**

In this chapter, the concept of structures is explained. You will learn how to
- define a new structure
- access the different members of a structure
- use struct variables as arguments and return values of functions
- use pointers to struct variables
- write to and read from files of structures

## 15.1 Definition

In chapter 7 the concept of arrays was introduced. These arrays are used to combine different variables into 1. Unfortunately, all variables in an array need to be of the same datatype. In real life, however, we often encounter a group of variables of different datatypes that belong together. Such a group of variables is called a record. To represent this real life situation, structures will be used.

A structure combines a set of variables of **different** datatypes into 1 variable.

Example:
we can use a structure to combine all product data. A product has a product number (type: int), a name or description (type: char *), an inventory level (type: int), a purchase price (type: float) and a retail price (type: float).

## 15.2 Defining a structure

A structure must be defined using the `struct` statement as follows:

```
struct <name>{
      datatype elem1;
      datatype elem2;
      ...
};
```

The statement above defines a new data type. Variables of that new datatype can be declared by:

```
struct <name> <variable_name>;
```

For instance, the definition of the structure "product" looks like:

```
struct product{
       int number;
       char description[30];
       int inventory;
       float purchase;
       float retail;
};
```

and the declaration of variables of this new datatype is done as follows:

```
struct product art1, art2;
```

or an array of variables of this new datatype:

```
struct product art[100];
```

`typedef` can be used in combination with `struct` to define and name a new data type. This new datatype can then be used to define structure variables directly as follows:

```
typedef struct product{
       int number;
       char description[30];
       int inventory;
       float purchase;
       float retail;
}Product;

Product art1, art2, art[100];
```

or:

```
typedef struct{
       int number;
       char description[30];
       int inventory;
       float purchase;
       float retail;
}Product;

Product art1, art2, art[100];
```

**Learning note**

Note that the newly defined `struct` type can only be used after its definition. Therefore, define structure variables always in the top section of the source code or in the header file if any.

## 15.3 Accessing structure members

Once the structure is defined and variables of the new datatype are declared, the different members of those variables need to be accessible. To access any member of a structure, the dot operator (.) is used. This dot operator is placed in between the variable name and the structure member name.

Example:

```
art1.number = 12;
strcpy(art[20].description, "plasma TV");
```

The individual structure members can be used like any other variable of that same type as can be seen in the examples above.

Also the structure variables themselves are regarded as ordinary variables. As a result, a structure variable can be used directly in an assignment which is not the case for array variables:

```
art[0] = art1;
```

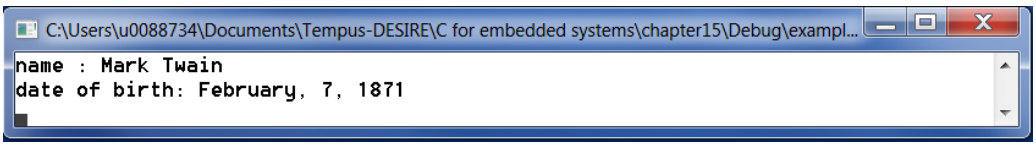
Initialization of a structure variable can also be done together with the declaration as follows:

```
Product art1={12, "plasma TV", 35, 245.50, 999.99};
```

## 15.4 Nested structures

Next to variables and arrays, structures can also contain other structures. The following example shows how nested structures can be used.

```
1   #include<stdio.h>
2
3   typedef struct
4   {
5      unsigned short day;
6      char month[16];
7      unsigned int year;
8   } Date;
9
10  typedef struct
11  {
12     char name[32];
13     Date DateOfBirth;
14  } Person;
15
16  int main(void)
17  {
18     Person p = { "Mark Twain", { 7, "February", 1871 } };
19     printf("name : %s\n", p.name);
20     printf("date of birth: %s, %u, %u\n", p.DateOfBirth.month,
                  p.DateOfBirth.day, p.DateOfBirth.year);
21     return 0;
22  }
```




**Code 74: usage of nested structures**

Note that the structure "`Date`" needs to be defined before it can be used in the structure "`Person`".

## 15.5 Structures and functions

Individual structure members as well as entire structures can be passed to a function as argument or returned from a function as function return value.

When an individual structure member or an entire structure is passed to a function as argument, it is passed by value, meaning that the structure value in the calling function cannot be modified from within the called function. The modifications done inside the called function can be returned to the calling function by the use of a function return value. This is illustrated in the example of Code 75 that shows the definition of the function `adapt`.

```
Product adapt(Product x)
{
	x.number += 1000;
	x.retail *= 1.2;
	return x;
}
```

**Code 75: example structures and functions**

Calling the function can be done as follows:

```
art[5] = adapt(art1);
```

Like for any other variable that is passed by value, the values of the structure members of `art1` are copied into the variable `x` which acts as a local variable in the function `adapt`. The return statement takes care of copying all members of `x` into `art[5]`.

## 15.6 Comparing 2 structures

C does not provide an equality operator that allows to directly compare 2 structures. Comparison can only be done by comparing the structure variables member by member.

The following example shows a function that can be used to compare two structures of the type `Product`:

```
int compare(Product x, Product y)
{
	if(x.number != y.number) return 0;
	if(x.inventory != y.inventory) return 0;
	if(x.purchase != y.purchase) return 0;
	if(x.retail != y.retail) return 0;
	return !strcmp(x.description, y.description);
}
```

## 15.7 Pointers to structures

Like for any other data type, pointers to structures can be declared as shown in following example:

```
Product a;
Product *pa;
pa = &a;
```

Where `pa` is declared to be a pointer to a variable of the type `Product`. The statement "`pa=&a;`" assigns the address of the structure variable `a` to the pointer `pa`.

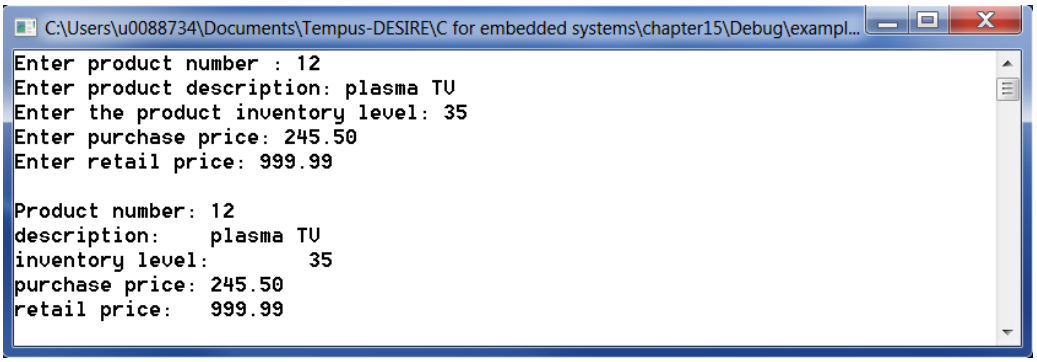
To access the individual structure members via the pointer variable, the `->` operator needs to be used:

```
pa->number = 1023;
strcpy(pa->description, "dvd");
```

Pointers to structures also allow to pass a structure to a function by reference as is illustrated in Code 76:

```
1   #include <stdio.h>
2
3   typedef struct {
4      int number;
5      char description[30];
6      int inventory;
7      float purchase;
8      float retail;
9   } Product;
10
11  void read(Product *);
12  void print(Product);
13
14  int main(void)
15  {
16     Product a;
17     read(&a);           //variable a is passed by reference
18     print(a);           //variable a is passed by value
19     return 0;
20  }
21
22  void read(Product *p)
23  {
24     printf("Enter product number : ");
25     scanf("%d%*c", &(p->number));
26
27     printf("Enter product description: ");
28     gets(p->description);
29
30     printf("Enter the product inventory level: ");
31     scanf("%d%*c", &(p->inventory));
32
33     printf("Enter purchase price: ");
34     scanf("%f%*c", &(p->purchase));
35
36     printf("Enter retail price: ");
37     scanf("%f%*c", &(p->retail));
38  }
39
```

```
40  void print(Product z)
41  {
42     printf("Product number:\t%d\n", z.number);
43     printf("description:\t%s\n", z.description);
44     printf("inventory level:\t%d\n", z.inventory);
45     printf("purchase price:\t%.2f\n", z.purchase);
46     printf("retail price:\t%.2f\n", z.retail);
47  }
```




**Code 76: passing structures to functions by reference**

**Common mistake**

Note the parenthesis around the combination `pointer->member` when an address operator (`&`) is used. Omitting these parenthesis is a commonly made mistake that results in accessing wrong memory locations.

## 15.8 Files of structures

Structures can be written to or read from a binary file as a whole using the functions `fread` and `fwrite` respectively. To determine the number of bytes to be read or written, the function `sizeof` is used:

```
struct Person pers;
fread(&pers, sizeof(pers), 1, fptr);
```

the function `fread` will read all data from 1 person from the file referred to by `fptr`, into the variable `pers`.

Similarly

```
struct Person person[20];
fread(person, sizeof(person[0]), 20, fptr)
```

will read all data from 20 persons from the file referred to by `fptr` and store them in an array of structures called `person`.

The usage of the `fwrite` function in combination with structures, is illustrated in Code 77. In this example, a file with phone numbers is created:

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #include "phone.h"
5
6   int readPhone(Phone *);
7
8   int main(void)
9   {
10     Phone phone;
11     FILE* fp;
12
13     if ((fp = fopen(FILENAME, "wb")) == NULL)
14     {
15            printf("\"%s\" cannot be opened\n", FILENAME);
16            exit(1);
17     }
18
19     while (readPhone(&phone))
20            fwrite(&phone, sizeof(Phone), 1, fp);
21
22     fclose(fp);
23     return 0;
24  }
25
26  int readPhone(Phone *pa)
27  {
28     printf("Enter name (<enter> to stop): ");
29     gets(pa->name);
30
31     if (pa->name[0] == '\0')
32            return 0;
33
34     printf("Enter phone number : ");
35     gets(pa->phoneNr);
36     return 1;
37  }
```

With phone.h containing:

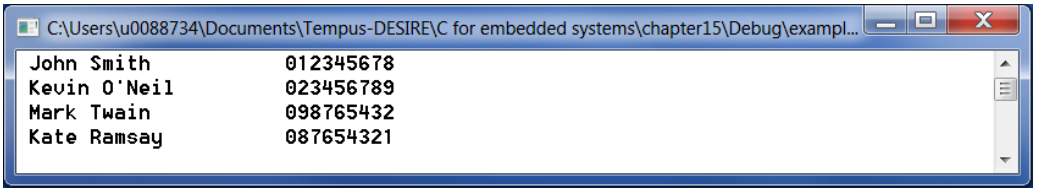
```
1   #ifndef _PHONE_H
2   #define _PHONE_H
3
4   #define FILENAME "phone.dat"
5
6   typedef struct {
7      char name[20];
8      char phoneNr[12];
9   } Phone;
10  #endif
```

**Code 77: writing structures to files**

The next example shows how to read the phone numbers back from the file and print them to the screen.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #include "phone.h"
5
6   void printPhone(Phone *);
7
```

```
8   int main(void)
9   {
10     Phone phone;
11     FILE* fp;
12     if ((fp = fopen(FILENAME, "rb")) == NULL)
13     {
14           printf("\"%s\" cannot be opened\n", FILENAME);
15           exit(1);
16     }
17
18     while (fread(&phone, sizeof(Phone), 1, fp) >0)
19           printPhone(&phone);
20
21     fclose(fp);
22     return 0;
23  }
24
25  void printPhone(Phone *pa)
26  {
27     printf(" %-20s %-12s\n", pa->name, pa->phoneNr);
28  }
```




**Code 78: reading structures from files**

Finally, we can combine the write and read operation into 1 program and add a menu to offer the user the possibility to choose what to do next:

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #include "phone.h"
5
6   char menu(void);
7   void add(FILE *);
8   void list(FILE *);
9   void search(FILE *);
10  void takeOut(FILE *);
11  void printPhone(Phone *, int);
12  void line(void);
13
14  int main(void)
15  {
16     FILE* fp;
17     int stop;
18
19     if ((fp = fopen(FILENAME, "r+b")) == NULL)
20     {
21           if ((fp = fopen(FILENAME, "w+b")) == NULL)
22           {
23                 printf("\"%s\" cannot be opened\n", FILENAME);
24                 exit(1);
25           }
26     }
27
28     stop = 0;
29
```

```
30    while (!stop)
31    {
32           switch (menu())
33           {
34           case 'e': stop = 1; break;
35           case 'a': add(fp); break;
36           case 'l': list(fp); break;
37           case 's': search(fp); break;
38           case 'r': takeOut(fp); break;
39           }
40    }
41    fclose(fp);
42    return 0;
43 }
44
45 char menu(void)
46 {
47    char s[16]; int i;
48    for (i = 0; i < MENU_SIZE; i++)
49    {
50           printf(" %s : %s\n", menutext[i][0], menutext[i][1]);
51    }
52    printf("Your choice: ");
53    gets(s);
54    return s[0];
55 }
56
57 void add(FILE *fp)
58 {
59    Phone a;
60    Phone *pa;
61    pa = &a;
62    memset(pa, '\0', sizeof(Phone)); // set all bytes in Phone to \0
63    printf("Name: ");
64    gets(pa->name);
65    printf("Phone Number: ");
66    gets(pa->phoneNr);
67    fseek(fp, 0, SEEK_END);
68    fwrite(pa, sizeof(Phone), 1, fp);
69 }
70
71 void list(FILE *fp)
72 {
73    Phone a;
74    Phone *pa;
75    int recnr = 0;
76    pa = &a;
77    line();
78    fseek(fp, 0, SEEK_SET);
79    while (fread(pa, sizeof(Phone), 1, fp) > 0)
80    {
81           // removed records with name[0]=='\0' cannot be printed
82           if (pa->name[0] != '\0')
83                  printPhone(pa, recnr);
84           recnr++;
85    }
86    line();
87 }
88
89 void search(FILE *fp)
90 {
91    …
92 }
93
94 void takeOut(FILE *fp)
95 {
96    // put all bytes to '\0' in the record you want to remove
97 }
98 void printPhone(Phone *pa, int recnr)
```

```
99  {
100    printf("%3d : %20s  %12s\n", recnr, pa->name, pa->phoneNr);
101 }
102
103 void line(void)
104 {
105    int i;
106    for (i = 0; i<50; i++)
107           printf("-");
108    printf("\n");
109 }
```

With phone.h containing:

```
1   #ifndef _PHONE_H
2   #define _PHONE_H
3
4   #define FILENAME "phone.dat"
5   #define NLEN 25
6   #define PLEN 20
7   #define MENU_SIZE 5
8
9   typedef struct
10  {
11     char name[NLEN];
12     char phoneNr[PLEN];
13  } Phone;
14  char* menutext[][2] = {{"a","add"},{"l","list"},{"s","search"},
                          {"r","remove"},{"e","end"}};
15  #endif
```

**Code 79: phone numbers program with menu**

## 15.9 Exercises

**15.1.** Write a program with functions:
- define a structure "`Person`" with 2 members: `name` and `firstname`
- declare an array of N persons (take N = 5 for instance)
- read the data of N persons
- print that data

write at least following functions:
- a function to read the data of 1 `Person`
- a function to print the data of 1 `Person`

The main program looks like:

```
#include <stdio.h>
#define N 5

typedef struct {
…
}Person;

void readPerson(…);
void printPerson(…);
```

```
int main(void)
{
      Person p[N];
      int i;
      printf("Enter %d names (first name and surname)\n", N);
      for(i=0; i < N; i++) readPerson(&p[i]);
      printf("The %d entered names are\n", N);
      for(i=0; i < N; i++) printPerson(p[i]);
      return 0;
}
```

**15.2.** Repeat exercise 15.1 with following additions:

- a structure `Date` with a day(`int`), a month(`string`) and a year(`int`)
- a function that reads a `Date`
- a function that prints a `Date`
- add a `dateOfBirth` to the structure `Person`. `dateOfBirth` must be of the type `Date`.
- the function to read a `Date` must be called in the function `readPerson`
- the function to print a `Date` must be called in the function `printPerson`

**15.3.** Repeat exercise 15.2 with addition of an `enrollmentDate` added to the structure `Person`. Change only the structure `Person` and use the existing functions to read and write a `Date` at the right place.

**15.4.** Define a structure `Address` with `streetAndNr`, `postalCode`, `town` and `phoneNr`. Define a structure `Student` with a `name`, `homeAddress` and `schoolAddress`. Write a program to test the structures defined.

Write a function that reads an `Address`:

```
readAddress(Address *p)
```

Both addresses are read with this function. Change the parameters passed on to the function to read in the correct type of address.

Ditto to print an `Address`:

```
printAddress( Address p)
```

**15.5.** Write a program that asks the user to enter the name and the home town of 3 persons. Name and home town are stored in a structure `Person`. The 3 persons are stored in an array. Afterwards, the program asks the user to enter a name and searches the town that person lives in.

Use the functions `readPerson` and `searchTown`
The function `searchTown` has 2 arguments:
- the array that needs to be searched
- a variable of the type `Person` that contains the name that needs to be searched. The `town` member of that variable needs to be filled with the town found by the function `searchTown`.

Reading the name of the person you want to search for is done in the `main` function. Printing the resulting town can also be done in the `main` function. If the name entered is not present in the array, an appropriate message needs to be printed.

```
Enter name: Smith
Enter town or city: Berkeley
Enter name: Minogue
Enter town or city: London
Enter name: O'Neil
Enter town or city: Dover
============================================================
Enter the name of the person you want to search for: Smith
This person lives in Berkeley
```

**15.6.** Write a program that reads name, age and salary of a chosen number of people and stores that information into a file. All data of 1 person is stored in a structure. Once the data of 1 person is read, the structure containing that data is written to a file at once. Afterwards, the data of the next person is read, … Make sure the names can contain spaces!

```
How many people do you want to enter? 3
Enter name: Smith
Enter age: 25
Enter salary: 1950
Enter name: Minogue
Enter age: 47
Enter salary: 6500
Enter name: O Neil
Enter age: 66
Enter salary: 2200
```

**15.7.** Write a program that reads the data from the file written in the previous exercise. Print the data to the screen as follows:

```
Name: Smith
Age: 25
Salary: 1950

Name: Minogue
Age: 47
Salary: 6500

Name: O Neil
Age: 66
Salary: 2200
```

**15.8.** Write a program that searches the age and wages of a person based upon a name entered by the user:

```
What's the name of the person you want to search for? Smith
Name: Smith
Age: 25
Salary: 1950
```

**15.9.** Now, we want to add the first name of all people in the file. Ask the first names and save them together with the already existing data in a new file. Print the list of data:

```
Enter the first name of Smith: Will
Enter the first name of Minogue: Kylie
Enter the first name of O Neil: Kate

following data was entered:
Name: Smith
First name: Will
Age: 25
Salary: 1950

Name: Minogue
First name: Kylie
Age: 47
Salary: 6500

Name: O Neil
First name: Kate
Age: 66
Salary: 2200
```

**15.10.** Complete the example of Code 79 in section 15.8 by writing the missing functions.

**15.11.** Repeat exercises 15.10 and add for each `Person` also street, number, postal code and town information.

**15.12.** Make a text file that contains:

Will Smith; 132 King Street; 2000; Berkeley
Kylie Minogue; 47 Jason Street; 2850; London
Kate O Neil; 65 York Street; 6547; Dover

or

Will Smith 132 King Street 2000 Berkeley
Kylie Minogue 47 Jason Street 2850 London
Kate O Neil 65 York Street 6547 Dover

write a program that reads 1 of these text files line by line and adds the data into the file of the previous exercise.

**15.13.** Repeat exercise 15.7 and add a function that prints the addresses ordered alphabetically.

Hint: define a two dimensional array with a record number for every person in the file:

| | |
|---|---|
| Smith | 1 |
| Minogue | 2 |
| O Neil | 3 |
| Allen | 4 |
| Donovan | 5 |

Now order this matrix:

| | |
|---|---|
| Allen | 4 |
| Donovan | 5 |
| Minogue | 2 |
| O Neil | 3 |
| Smith | 1 |

Read the records from the file in the order specified by the matrix: first record number 4, then record number 5, …

# 16 Command line arguments

**Objectives**

In this chapter you will learn how to pass arguments to your programs from the command line.

## 16.1 argc, argv[]

In C, it is possible to pass arguments to your program from the command line. These arguments are handled by including the parameters `int argc` and `char * argv[]` into the parameter list of the main function:

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

The parameter `argc`, short for argument counter, contains the number of arguments passed from the command line including the program name. The argument vector, `argv[]`, contains an array of char pointers, pointing to the first, second, … argument respectively.

Code 80 demonstrates the usage of the command line arguments.

```
1   #include <stdio.h>
2
3   int main(int argc, char *argv[])
4   {
5      int i;
6      printf("argc = %d\n", argc);
7
8      for (i = 0; i < argc; i++)
9      {
10           printf("argv[%d] = %s\n", i, argv[i]);
11     }
12     return 0;
13  }
```

**Code 80: command line arguments**

Assuming that after compilation, the executable file for above program is called `myprogram`, it produces following result:

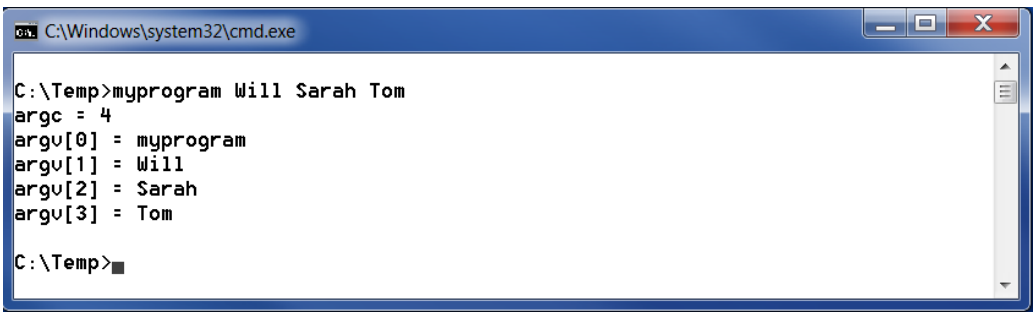



**Figure 51: command line arguments**

Note that the content of `argv[]` is depending on the arguments entered by the user. If, for instance, the user enters too little arguments, a run time error might occur. This can be avoided by verifying the number of arguments entered before any further manipulations of the command line arguments:

```
if (argc<3)
{
       printf("too little arguments");
       exit(1);
}
```

**Common mistake**

Since `argv[]` is an array of char pointers, the different array elements are pointers to strings! Saving a string into a variable must be done using the function `strcpy`! Using a simple assignment operator is a commonly made mistake.

**Common mistake**

If a number is to be passed to the program from the command line, the corresponding string `argv[i]` must be converted into a number before using it! Omitting the string to integer conversion is a commonly made mistake. The conversion can be carried out with the function `atoi()`.

## 16.2 Exercises

**16.1.** Write a program with name `mycopy.exe` that expects 2 command line arguments and that copies one file to another.

Example:

```
mycopy text.txt copy.txt
```

copies the content of text.txt into the file copy.txt.

**16.2.** Write a program with command line arguments that compares two text files. If the files are identical, the program will print an appropriate message. If not, all lines containing differences must be printed to the screen with their line numbers. If the files differ in more than 10 lines, the program can stop printing after the $10^{th}$ line.

extra: try to redirect the screen output to a text file without changing the source code.

**16.3.** Write a program with command line arguments that calculates the age (in number of years) of a number of people based upon their date of birth and the current date.

- make a text file containing a number of names with their corresponding dates of birth. Make sure names and dates of birth are always written in the same way in the text file.
- write a function "`read_person`" to read the data of 1 person from the file and save that data into a `struct` of the type "`Person`". This `struct` contains a name field and a field to save the date of birth, which is a `struct` of the type "`Date`".
- call the program from the command line with the current date and the text file as arguments
- write a function "`calculate`" that calculates the age of a person in years, months and days. This function also determines which person is the oldest and which person has the longest name.
- printing the ages is done in the `main` function. Print also the names of the oldest person and of the person with the longest name.

```
age_calc ages.txt 01 12 2014

the ages are:
Wilbur is 18 years old
Tom is 18 years old
Sarah is 15 years old

Tom is the oldest of the 3 persons in the file.
The person with the longest name is: Wilbur
```

The text file ages.txt contains:

```
Wilbur
01 03 1996
Tom
25 12 1995
Sarah
06 06 1999
```

# 17 Dynamic memory allocation

**Objectives**

In this chapter, the concept of allocating and freeing memory at run time is explained.
Following functions will be handled:

- `malloc`
- `free`
- `calloc`
- `realloc`

## 17.1 Introduction

So far, memory allocations were done explicitly in the source code by indicating the variable type and, for arrays and structures, also the variable length. As a result, the programmer needs to know in advance how much memory will be needed. If, for instance, there is some uncertainty on the maximal number of array elements needed for program execution, the programmer would be forced to choose a large enough number even if for 90% of the time the array will contain only 2 valid elements.

Therefore, dynamic memory allocation is used. It allows to allocate extra memory while the program is running and to free that memory as soon as possible.

The functions used to accomplish this are all defined in "`stdlib.h`" and will be treated in the following sections.

## 17.2 The function `malloc`

The function `malloc`, with function declaration:

```
void * malloc(size_t size);
```

allocates `size` bytes in memory at runtime. If the memory allocation went well, a pointer to the newly allocated memory is returned. Since `malloc` can be used to allocate memory for all types of data, the returned pointer is of the type `void *`. Hence, type casting to the wanted data type will be needed. In case of an error during the memory allocation, NULL is returned instead.

To determine the number of bytes needed, the function `sizeof` can be used:

```
int *p;
p = (int *) malloc(sizeof(int));
```

**Common mistake**

The only reference to the dynamically created memory block is the pointer returned by the `malloc` function. Reusing that pointer for other purposes without freeing the memory first is a common mistake that leads to lost objects like in the code below:

```
char * p;
p = (char *) malloc(32);
strcpy(p, "lost");
p = (char *) malloc(32);
```

After copying the string "lost" into the first block of memory, the pointer `p` is altered to point to yet another newly created memory block. Since the memory containing the string "lost" has no name associated with it, it is now no longer possible to retrieve it!

## 17.3 The function `free`

Once the dynamically allocated memory is no longer needed, it can be freed using the function `free` as follows:

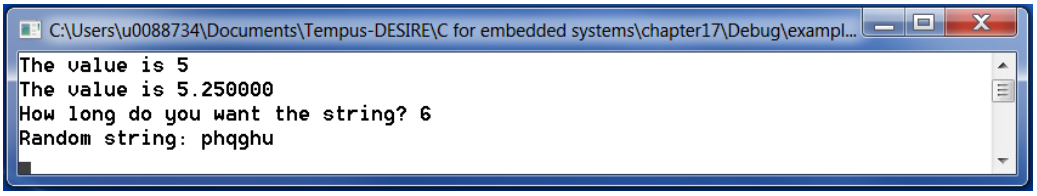
```
void free(void * ptr);
```

where `ptr` points to the memory block that needs to be released. If `ptr` is a `NULL` pointer, nothing will happen.

The usage of the functions `malloc` and `free` is demonstrated in Code 81:

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(void)
5   {
6      int* p;
7      double* q;
8      int i, n;
9      char * buffer;
10
11     p = (int*)malloc(sizeof(int));
12     *p = 5;
13     printf("The value is %d\n", *p);
14     free(p);
15
16     q = (double*)malloc(sizeof(double));
17     *q = 5.25;
18     printf("The value is %lf\n", *q);
19     free(q);
20
21     printf("How long do you want the string? ");
22     scanf("%d%*c", &i);
23     buffer = (char*)malloc(i + 1); //+1 for the null byte at the end
24
25     if (buffer == NULL)
26           exit(1);
```

```
27     for (n = 0; n<i; n++)
28           buffer[n] = rand() % 26 + 'a';
29
30     buffer[i] = '\0';
31     printf("Random string: %s\n", buffer);
32     free(buffer);
33
34     return 0;
35  }
```




**Code 81: `malloc` and `free` example**

## 17.4 The function `realloc`

If the size of a previously dynamically allocated memory block needs to be changed, the function `realloc` can be used:

```
void * realloc(void *ptr, size_t size);
```

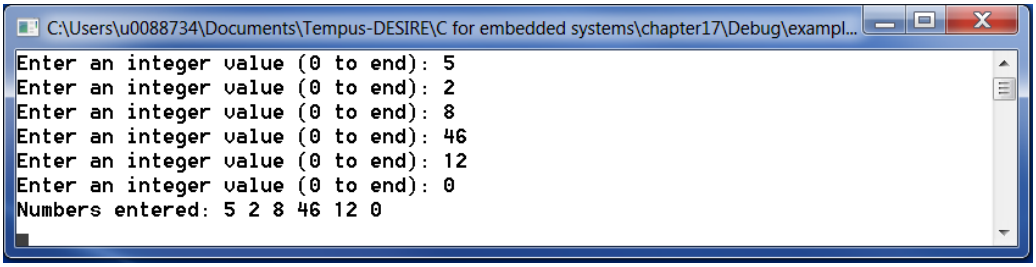
The pointer `ptr` points to the original memory block. If this pointer is `NULL`, the function `realloc` acts like the function `malloc`. The new size, in bytes, can be specified with the `size` argument. If `size` equals zero, the memory `ptr` points to, is freed and the function returns `NULL`.

If sufficient space is available to expand the original memory block to the newly wanted size, the additional memory is allocated and the function returns the same pointer `ptr`. If not, a new block of `size` bytes is allocated, the content of the original memory block is copied into the new block and the original memory block is freed. The function now returns a pointer to the newly created block. In case the reallocation failed, the value `NULL` is returned.

Code 82 shows an example where the function `realloc` is used:

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(void)
5   {
6      int input, n;
7      int count = 0;
8      int *numbers = NULL;
9      int *more_numbers;
10
11     do
12     {
13        printf("Enter an integer value (0 to end): ");
14        scanf("%d%*c", &input);
15        count++;
```

```
16
17          more_numbers = (int*)realloc(numbers, count * sizeof(int));
18
19          if (more_numbers != NULL)
20          {
21              numbers = more_numbers;
22              numbers[count - 1] = input;
23          }
24          else
25          {
26              free(numbers);
27              puts("Error (re)allocating memory");
28              exit(1);
29          }
30      } while (input != 0);
31
32      printf("Numbers entered: ");
33      for (n = 0; n<count; n++)
34         printf("%d ", numbers[n]);
35
36      printf("\n");
37      free(numbers);
38      return 0;
39  }
```




**Code 82: usage of `realloc`**

## 17.5 The function `calloc`

The function `calloc`, with function declaration:

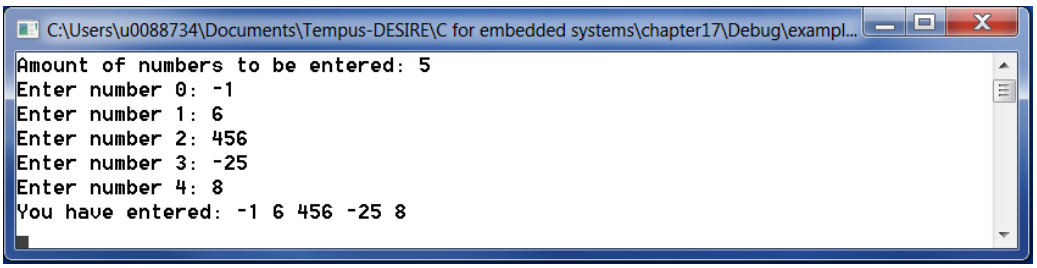
```
void * calloc(size_t num, size_t size);
```

allocates memory for an array of `num` elements of `size` bytes each and initializes all elements to 0. The return value is again a pointer to the allocated memory block or `NULL` if the memory allocation failed.

In the example below, the function `calloc` is used to read and print an array of integers:

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(void)
5   {
6      int i, n;
7      int* pData;
8      printf("Amount of numbers to be entered: ");
9      scanf("%d%*c", &i);
10     pData = (int*)calloc(i, sizeof(int));
```

```
11
12      if (pData == NULL)
13              exit(1);
14
15      for (n = 0; n<i; n++)
16      {
17              printf("Enter number %d: ", n);
18              scanf("%d%*c", pData+n);
19      }
20
21      printf("You have entered: ");
22      for (n = 0; n<i; n++)
23              printf("%d ", pData[n]);
24
25      printf("\n");
26      free(pData);
27      return 0;
28  }
```

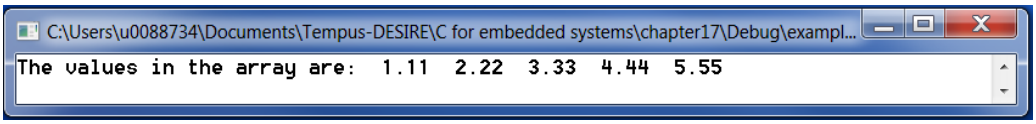



**Code 83: usage of `calloc`**

## 17.6 Dynamic arrays

As explained in chapter 7, arrays are stored in contiguous memory locations. Therefore, an array of `num` elements of `size` bytes each will occupy a memory block of `num*size` bytes. As a result, allocating memory for such an array at runtime can be done using the function `malloc` as illustrated in the next example:

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(void)
5   {
6      double* p;
7      int i;
8      p = (double*)malloc(5 * sizeof(double)); // allocate memory for
9                                                  an array of 5 doubles
10     *p = 1.11;
11     *(p + 1) = 2.22;
12     *(p + 2) = 3.33;
13     *(p + 3) = 4.44;
14     *(p + 4) = 5.55;
15
16     printf("The values in the array are: ");
17     for (i = 0; i<5; i++)
18             printf("%5.2f ", p[i]);
19
20     printf("\n");
21     free(p);
22     return 0;
23  }
```

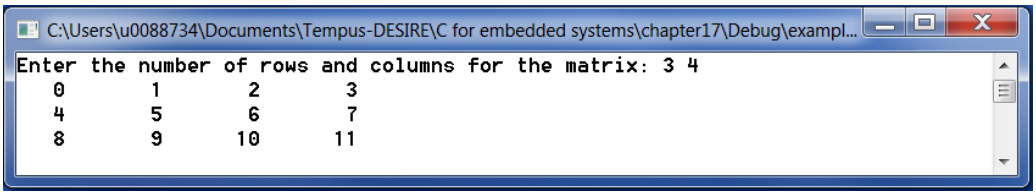



**Code 84: dynamic arrays**

Also multidimensional arrays can be dynamically allocated using `malloc`. Remember that an array of r rows and k columns, is stored in memory in r*k contiguous memory locations.

Example:
Write a program that asks the user to enter the number of rows and columns needed, that dynamically creates the requested array, fills it with data and prints that data.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   void fillMatrix(int *, int, int);
4   void printMatrix(int *, int, int);
5
6   int main(void)
7   {
8      int rows, columns;
9      int *p;
10     printf("Enter the number of rows and columns for the matrix: ");
11     scanf("%d%d%*c", &rows, &columns);
12
13     p = (int *)malloc(rows*columns*sizeof(int));
14     fillMatrix(p, rows, columns);
15     printMatrix(p, rows, columns);
16     return 0;
17  }
18
19  void printMatrix(int *m, int r, int k)
20  {
21     int i;
22     for (i = 0; i < (r*k); i++)
23     {
24            printf("%4d\t", *(m + i));
25            if ((i + 1) % k == 0)      //print only k elements per line
26                   printf("\n");
27     }
28  }
29
30  void fillMatrix(int *m, int r, int k)
31  {
32     int i;
33     for (i = 0; i < (r*k); i++)
34     {
35            *(m + i) = i;
36     }
37  }
```




**Code 85: dynamic multidimensional array**

## 17.7 Exercises

**17.1.** Write a program that sums all elements of the matrix diagonals and prints the maximal array element. The matrix dimensions are chosen by the user.

```
What is the matrix dimension? 3
Enter the matrix
1  4  2
2  5  1
2  4  8

The diagonal top left to bottom right sums up to 14
The diagonal bottom left to top right sums up to 9
The maximum number in the matrix is 8
```

Use a function `readArray` to read the content of the matrix and a function `calculate` to calculate the diagonals and the maximum number. Printing can be done in the `main` function.

**17.2.** Write a program that asks the user to enter the wanted number of rows and columns, creates the matrix dynamically, fills it and prints the content. Use a separate function to fill the matrix and one to print the matrix. Fill every matrix element with `(row+1)*(column + 1)`

```
Enter the number of rows and columns for the matrix: 2 4

The matrix contains following elements:
1  2  3  4
2  4  6  8
```

**17.3.** Write a program with name "clients" that:

- reads customer numbers and corresponding customer names and stores them in an array of `structs`.
- reads customer numbers and corresponding customer addresses and stores them in a second array of `structs`. Use a different `struct`.
- can be called from the command line with the size of the arrays as argument (ex: clients 3)
- uses arrays of the correct size (use malloc)
- prints the customer data

use a separate function to read the data and one to print the data.

```
clients 3

Enter a list of 3 customer numbers and corresponding names:
AB
John Smith
AC
Tom Black
AD
Sarah White

Enter a list of 3 customer numbers and corresponding
addresses (the customer numbers must be identical to
the ones above but can be entered in an arbitrary order.)
AD
London
AB
Paris
AC
Brussels

All customer data:
AB  John Smith        Paris
AC  Tom Black         Brussels
AD  Sarah White       London
```

# 18 Dynamic data structures

**Objectives**

In this chapter, you will learn more about linked data structures. You will learn how to create and manipulate linked lists, stacks and queues.

## 18.1 Introduction

In chapter 15, we introduced structures to group variables of different types together to emphasis the cohesion between them. In this chapter, we will use structures to create a special kind of data. To do so, we first need to define a self-referential element. This is a structure that has a pointer to a structure of the same type as one of its members as illustrated in the example below:

```
typedef struct node
{
	char name[64];
	struct node *next;
} Node;
```

**Code 86: self-referential structure**

Combining these self-referential structures with dynamic memory allocations allows to create a data structure that can grow and shrink at execution time. Such data structures are called dynamic data structures.

Different types of dynamic data structures exist in C. In the next chapters (single-)linked lists, queues and stacks will be treated. Next to linear data structures, also binary trees are a commonly used examples of dynamic data structures. Unfortunately, binary trees fall beyond the scope of this course.

## 18.2 Linked lists

### 18.2.1 Definition

A linked list is a dynamic data structure that consist of a number of nodes. Each node is a self-referential structure that contains one or more data fields and one pointer member that is used to point to the next node. As a result a chain of nodes is built. The first node of the list is accessed via a head pointer. The last node of the list does not need to point to anything, therefore, the link pointer of this node is set to NULL. Figure 52 shows an example of a linked list with one data field per node.

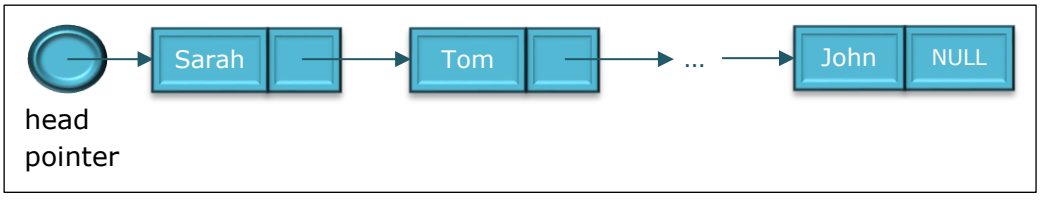



**Figure 52: linked list**

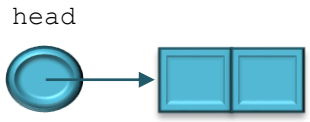
### 18.2.2 Creating a single-linked list

As node element, we will use the self-referential structure of Code 86. First we start by creating a head pointer. This is a pointer that will point to a node of the list hence it needs to be of the type `Node *`:

```
Node * head = NULL;
```


To create a first node in the list, memory needs to be allocated using `malloc` with `head` pointing to the newly allocated memory block:

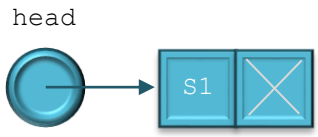
```
head = (Node*)malloc(sizeof(Node));
```




The only thing left to do now is assigning data to that first node:

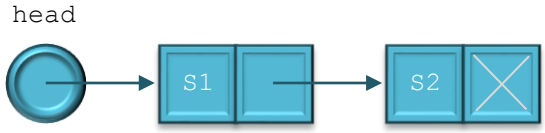
```
printf("Enter a name: "); gets(s);
strcpy(head->name, s);
head->next = NULL;
```

resulting in:




The second node of the list is created using the same steps. This time the newly allocated memory needs to be accessible from the next field of the first node resulting in:

```
head->next = (Node*)malloc(sizeof(Node));
printf("Enter a name: "); gets(s);
strcpy(head->next->name, s);
head->next->next = NULL;
```




Combining all pieces of code shown above and adding the creation of a third node in the list results in the C code of Code 87:

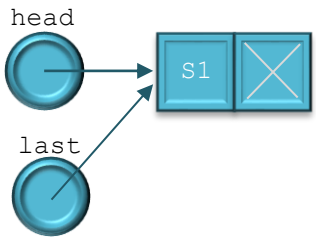
```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4
```

```
5   typedef struct node
6   {
7      char name[64];
8      struct node *next;
9   } Node;
10
11  void printList(Node *ptr);
12
13  int main(void)
14  {
15     Node *head;
16     char s[64];
17     // create first node
18     head = (Node*)malloc(sizeof(Node));
19     head->next = NULL;
20
21     // read a name
22     printf("Enter a name: ");
23     gets(s);
24
25     // copy name to first node
26     strcpy(head->name, s);
27
28     // print current list
29     printList(head);
30
31     // create second node
32     head->next = (Node*)malloc(sizeof(Node));
33     head->next->next = NULL;
34
35     // read a name
36     printf("Enter a name: ");
37     gets(s);
38
39     // copy name to second node
40     strcpy(head->next->name, s);
41
42     // print current list
43     printList(head);
44
45     // create third node
46     head->next->next = (Node*)malloc(sizeof(Node));
47     head->next->next->next = NULL;
48
49     // read a name
50     printf("Enter a name: ");
51     gets(s);
52
53     // copy name to third node
54     strcpy(head->next->next->name, s);
55
56     // print current list
57     printList(head);
58     return 0;
59  }
60
61  void printList(Node *h)
62  {
63     printf("The list contains:");
64     while (h != NULL)   //if h==NULL the list is finished
65     {
66           printf("%s ", h->name);
67           h = h->next; //move pointer h one node further
68     }
69     printf("\n");
70  }
```

**Code 87: creation of a single-linked list**

In the above example, addition of a new element requires running through the full list. This is very unpractical. Therefore, we will rewrite the previous example using a pointer that always points to the last element of the list:

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4
5   typedef struct node
6   {
7      char name[64];
8      struct node *next;
9   } Node;
10
11  void printList(Node *ptr);
12
13  int main(void)
14  {
15     Node *head;
16     Node *last;
17     char s[64];
18
19     // create first node
20     head = (Node*)malloc(sizeof(Node));
21     last = head;
22     last->next = NULL;
23     printf("Enter a name: ");
24     gets(s);
25     strcpy(last->name, s);
26
27     // print current list
28     printList(head);
29
30     // create second node
31     last->next = (Node*)malloc(sizeof(Node));
32     last = last->next;
33     last->next = NULL;
34     printf("Enter a name: ");
35     gets(s);
36     strcpy(last->name, s);
37     printList(head);
38
39     // create third node
40     last->next = (Node*)malloc(sizeof(Node));
41     last = last->next;
42     last->next = NULL;
43     printf("Enter a name: ");
44     gets(s);
45     strcpy(last->name, s);
46     printList(head);
47     return 0;
48  }
49
50  void printList(Node *h)
51  {
52     printf("The list contains:");
53     while (h != NULL)
54     {
55            printf("%s ", h->name);
56            h = h->next;
57     }
58     printf("\n");
59  }
```

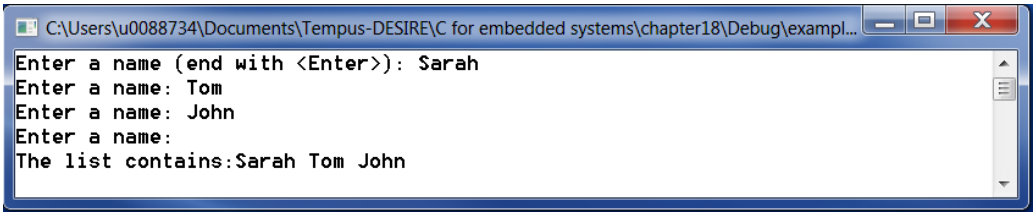



**Code 88: creation of a single-linked list (improved code)**

Finally, we can introduce loops containing the instructions that are repeated for every extra node creation:

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4
5   typedef struct node
6   {
7      char name[64];
8      struct node *next;
9   } Node;
10
11  void addNode(char *, Node **);
12  void printList(Node *ptr);
13
14  int main(void)
15  {
16     Node *head = NULL;
17     char s[64];
18
19     // read a name
20     printf("Enter a name (end with <Enter>): ");
21     gets(s);
22
23     // while there is input...
24     while (s[0])
25     {
26            // create a new node and insert into the list
27            addNode(s, &head);
28
29            // read next name
30            printf("Enter a name: ");
31            gets(s);
32     }
33     printList(head);
34     return 0;
35  }
36
37  void addNode(char *s, Node **h)
38  {
39     Node *new;
40     Node *temp;
41
42     // create a new node
43     new = (Node*)malloc(sizeof(Node));
44     new->next = NULL;
45
46     // copy s to the new node
47     strcpy(new->name, s);
48
49     // if there is no starting node yet, create one
50     if (*h == NULL)
51     {
52            *h = new;
53     }
54     // if there is a starting node, insert new node at end of list
55     else
56     {
57            temp = *h;
58
59            while (temp->next != NULL)    // go to end of list
60                   temp = temp->next;
61
62            temp->next = new;            // insert the new node
63     }
64  }
```

```
65  void printList(Node *h)
66  {
67     printf("The list contains:");
68     while (h != NULL)
69     {
70            printf("%s ", h->name);
71            h = h->next;
72     }
73     printf("\n");
74  }
```

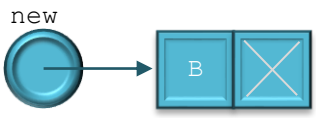



**Code 89: creation of a single-linked list using loops**

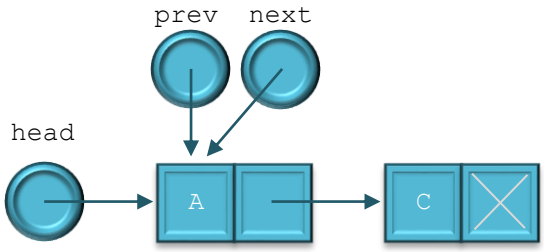
### 18.2.3 Insertion of a new node in a single-linked list

Suppose we build a list where all elements are ordered alphabetically. Insertion of a new element needs to be done in the correct place respecting the alphabetic ordering of the list. To this end, following steps are carried out:

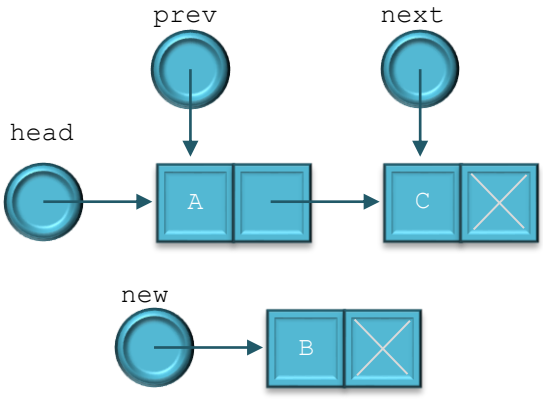
1. Creation of a new node:



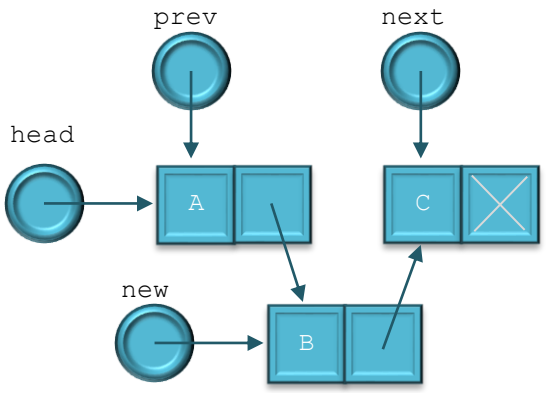

2. Initialization of the pointers "`prev`" and "`next`":




3. Positioning the pointers "`prev`" and "`next`" to the correct list elements



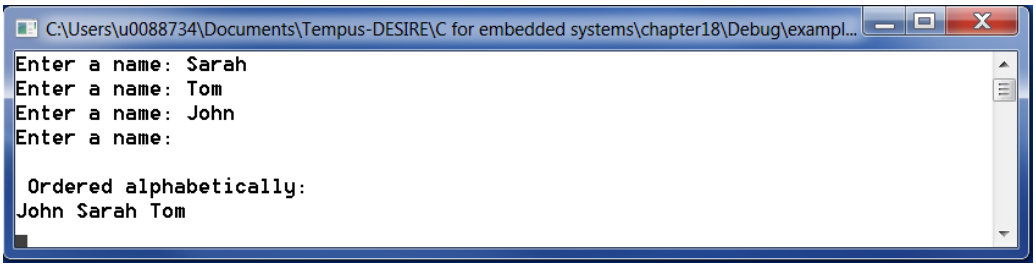
4. Insertion of the new element in between the elements pointed to by "prev" and "next"




Following code shows how a list can be used to alphabetize:

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4
5   typedef struct node
6   {
7      char name[64];
8      struct node *next;
9   } Node;
10
11  void insertNode(char s[], Node **ptr);
12  void printList(Node *ptr);
13
14  int main(void)
15  {
16     Node* head = NULL;
17     char s[64];
18
19     // read a name
20     printf("Enter a name: ");
21     gets(s);
22
23     // while still input...
24     while (s[0])
25     {
26            // create new node and insert in list (alphabetically)
27            insertNode(s, &head);
28            printf("Enter a name: ");
29            gets(s);
30     }
31     // print list
32     printList(head);
33     return 0;
34  }
35
36  void insertNode(char s[], Node **h)
37  {
38     Node *temp;
39     Node *prev;
40     Node *new;
41
42     // create new node
43     new = (Node*)malloc(sizeof(Node));
44     new->next = NULL;
45     strcpy(new->name, s);
46     // if list is empty, first node = new node
```

```
47      if (*h == NULL)
48      {
49              *h = new;
50      }
51      else   // if list is not empty, insert node in correct place
52      {
53              // if name new node<name first node, first node=new node
54              if (strcmp(s, (*h)->name) < 0)
55              {
56                      new->next = *h;
57                      *h = new;
58              }
59              else
60              {
61                      // initialize pointers prev and temp
62                      prev = *h;
63                      temp = prev;
64
65                      // while name new node > name current node and
                           list not empty, go to next node
66                      while (temp != NULL && strcmp(s, temp->name) >= 0)
67                      {
68                              prev = temp;
69                              temp = temp->next;
70                      }
71                      // adjust pointers to insert node
72                      new->next = temp;
73                      prev->next = new;
74              }
75      }
76  }
77
78  void printList(Node *h)
79  {
80     printf("\n Ordered alphabetically:\n");
81     while (h != NULL)
82     {
83             printf("%s ", h->name);
84             h = h->next;
85     }
86     printf("\n");
87  }
```

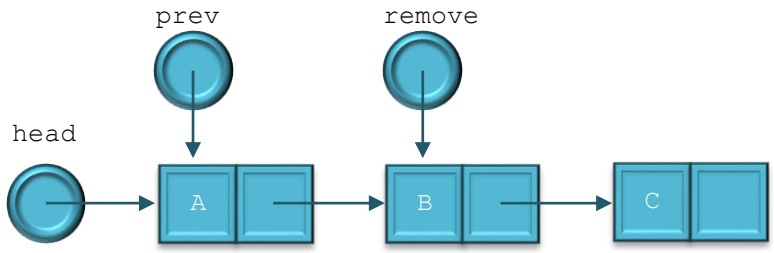



**Code 90: using a linked list to order alphabetically**

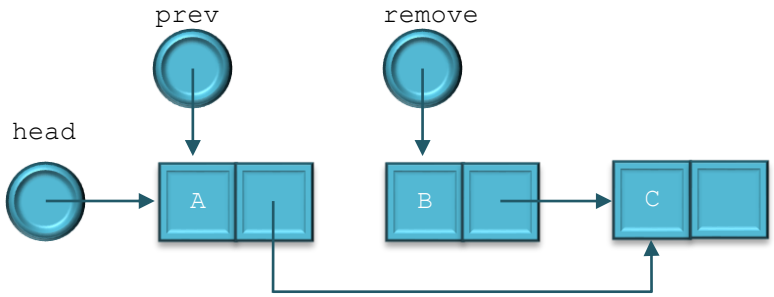
### 18.2.4 Removal of a node in a single-linked list

Removal of a specific node of a linked list can be done using following steps:

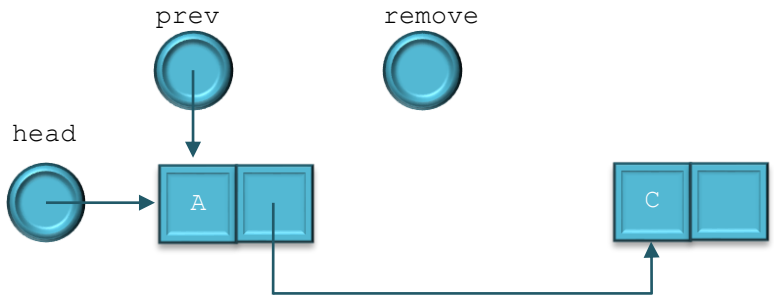
1. find the node to be removed and the node preceding that one:




2. change the next pointer of the preceding node:




3. remove the node (`free(remove)`):



### 18.2.5 Double-linked list

Traversal of a single-linked list can only be done in one way. As a result, a pointer to the previous node is needed for simple operations like insertion and deletion of a node. To overcome this problem, we could add a link to the previous element in each node resulting in following node structure:

```
typedef struct node
{
       char name[64];
       struct node *next;
       struct node *prev;
} Node;
```

Linking different elements of above node type together results in a double-linked list as illustrated in Figure 53.

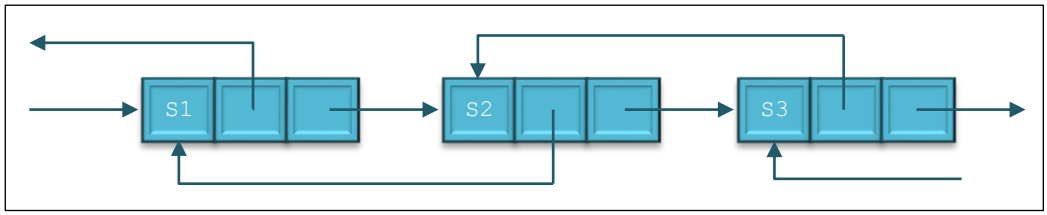



**Figure 53: double-linked list**

### 18.2.6 Circular linked list

A circular linked list is a linked list in which the next pointer of the last node points to the first node. Circular linked lists can be both single-linked and double-linked. Figure 54 shows an example of a circular single-linked list.

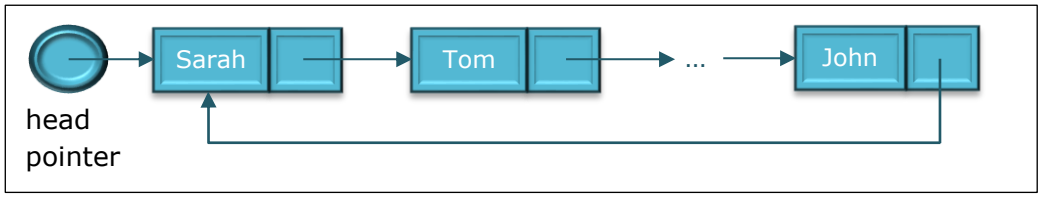



**Figure 54: circular single-linked list**

### 18.2.7 Stack

A stack is a linear list that can only be accessed from its top. Adding and removing nodes can be done only at one side of the list resulting in a Last In First Out or LIFO behavior. The top element in the stack is referenced via a stack pointer (stackPtr) as illustrated in Figure 55:

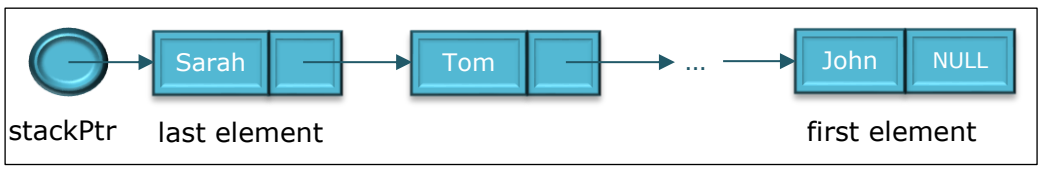



**Figure 55: Stack**

### 18.2.8 Queue

A queue is another type of linear list. Nodes can only be added at the tail of the queue and removed from the head of the queue, resulting in a FIFO or First In First Out behavior. In this case a head pointer and a tail pointer will be needed to refer to the 2 access points of the queue as can be seen in Figure 56:

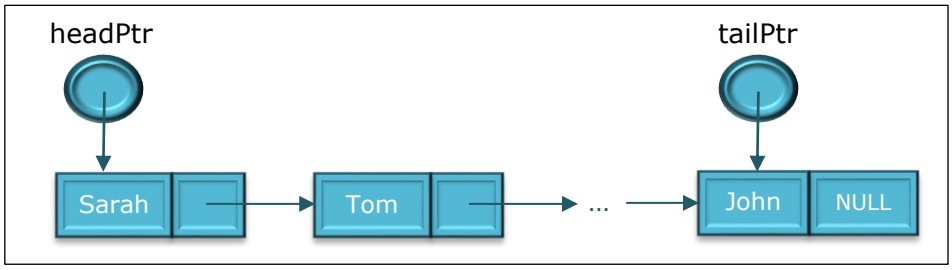



**Figure 56: queue**

## 18.3 Exercises

**18.1.** Write a program that allows to build and adapt an alphabetically ordered single-linked list with a command interpreter. Part of the code is written below. Only the missing functions need to be programmed.

```
#include <stdio.h>
#include <string.h> // needed for strcpy() and strcmp()
#include <stdlib.h> // needed for malloc(), free()
#define STRLEN 64

typedef struct node
{
    char name[STRLEN];
    struct node *next;
} Node;

void showHelp(void);
void add(char *, Node **);
void removeElement(char *, Node **);
int isItem(char *, Node *);
int countItems(Node *);
void removeList(Node **);
void showList(Node *);
char * trim(char *);

int main(void)
{
    char instruction[STRLEN];
    Node *head = NULL;
    char *pi; // pi = pointer to the instruction
    printf("Test program for a single-linked list.\n");
    printf("\nEnter an instruction (h = help)\n");
    printf("\n> ");
```

```
       memset(instruction, '\0', STRLEN);
       gets(instruction);
       pi = trim(instruction);

       while (*pi != 'q')
       {
              switch (*pi)
              {
              case 'i': add(trim(pi + 1), &head);
                     break;
              case 'd': removeElement(trim(pi + 1), &head);
                     break;
              case 'f': if (isItem(trim(pi + 1), head))
                           printf("\"%s\" is in the list.\n",
                                  trim(pi+1));
                        else
                           printf("\"%s\" is NOT in the list.\n",
                                  trim(pi+1));
                     break;
              case 'l': showList(head);
                     break;
              case 'n': printf(" Number of list items: %d\n",
                                countItems(head));
                     break;
              case 'r': removeList(&head);
                     break;
              case 'h': showHelp();
              case 'q': break;
              default: printf(" Unknown instruction (h = help)\n");
              }
              printf("\n> ");
              memset(instruction, '\0', STRLEN);
              gets(instruction);
              pi = trim(instruction);
       }

       removeList(&head);
       return 0;
  }

void showHelp(void)
{
       printf("i <string> : inserts the element in <string>
                            alphabetically into the list\n");
       printf("d <string> : removes the element in <string> from the
                            list\n");
       printf("f <string> : searches the list and returns if the string
                            is in the list or not.\n");
       printf("l : shows the full list\n");
       printf("n : returns the number of items in the list\n");
       printf("r : removes the full list\n");
       printf("h : shows the help menu (this list).\n");
       printf("q : end of the program (first remove the list)\n");
}

void add(char *s, Node **b)
{
       printf("This function inserts \"%s\" (alphabetically) into the
                list\n", s);
}
```

```
void removeElement(char *s, Node **b)
{
	printf("This function removes \"%s\" from the list\n", s);
}

int isItem(char *s, Node *b)
{
	printf("This function searches \"%s\" in the list\n", s);
	return …;
}

int countItems(Node *b)
{
	printf("This function returns the number of items in the
	list\n”);
	return …;
}

void removeList(Node **b)
{
	Node * p = *b;
	while (p != NULL)
	{
		*b = p->next;
		free(p);
		p = *b;
	}
}

void showList(Node *b)
{
	if (b == NULL)
	{
		printf("The list is EMPTY\n");
	}
	else
	{
		printf("The list:\n");
		while (b != NULL)
		{
			printf("%s ", b->name);
			b = b->next;
		}
	}
	printf("\n\n");
}

char * trim(char *s)
{
	while (*s == ' ') s++;
	return s;
}
```

Remark: the function trim is not absolutely needed for this exercise. It removes extra spaces at the beginning of an instruction or string.

**18.2.** Modify exercise 18.1 by using a **double-linked list**. The Node structure is defined as follows:

```
typedef struct node
{
   char name[64];
   struct node * previous;
   struct node *next;
} Node;
```

Use a head pointer and an end pointer that points to the last element of the list. Add also a function that prints the list in reverse order.

**18.3.** Write a program with a command interpreter that builds and manipulates a **stack**.
The commands you need to support are:

```
p <string>: add a string to the stack (push)
d: remove an element from the stack and print it (pop)
l: show the full stack
q: end of the program. (remove the stack first)
```

Hint: declare a pointer to the last element of the stack (stackpointer)

**18.4.** Write a program with a command interpreter that builds and manipulates a **queue**.
The commands you need to support are:

```
a <string>: add a string to the queue
d: remove an element from the queue and print it
l: show the full queue
q: end of the program. (remove the queue first)
```

Hint: declare a pointer to the first element of the queue (at this side elements can be removed) and a pointer to the last element of the queue (at this side elements can be added).

**18.5.** Modify exercise 18.1 using a **single-linked circular list**.

**18.6.** Modify exercise 18.2 using a **double-linked circular list**.

**18.7.** Add following commands to exercise 18.1 or 18.2.

s <filename> : saves the names in the list to a file
o <filename> : reads names from a file and inserts them into a list

**18.8.** The problem of Josephus.
Suppose N people decide to commit suicide together. To do so, they stand in a circle and every M people, 1 person is eliminated. What is the order of elimination?

```
Enter the number of people in the circle: 7
Enter the number M: 4

The order of elimination is: 4 1 6 5 7 3 2
```

Hint: use a circular list and count each time 4 items.

# Literature


- Teach yourself programming in 21 days; Peter Aitken, Bradley L. Jones; Sams Publishing; ISBN 0-672-30736-7
- C How to Program seventh edition, international edition; Harvey Deitel, Paul Deitel; Pearson Education; ISBN 0-273-77684-3
- De programmeertaal C (4de vernieuwde editie); Al Kelley, Ira Pohl; Pearson Education; ISBN 978-90-430-1669-8

# Attachments

## 1. Visual Studio Express 2013 for Desktop

In this course, we will use Visual Studio Express 2013 for Desktop as development environment.

Visual Studio works with solutions and projects.
One solution can contain one or more projects and every project can contain one or more source files, header files, text files, …

In figure 7, you see an example of a solution called "`structures`" that includes 6 projects. The project "`DoubleLinkedList`" contains a header file "`Header.h`" and a source file "`DoubleLinkedList.c`"

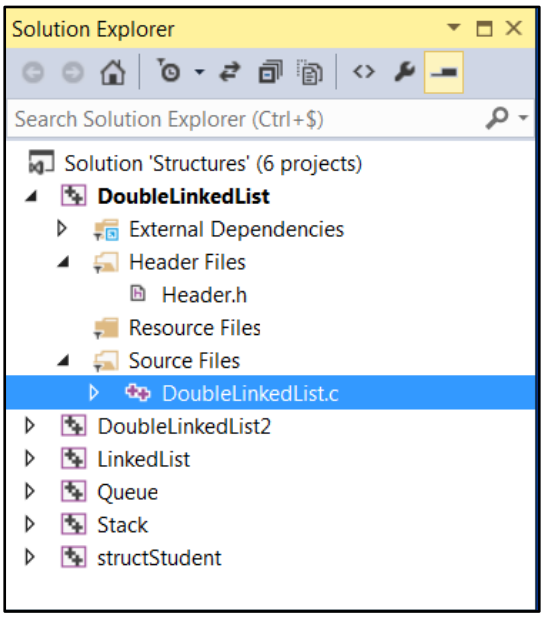



**Figure 57: solution with 6 projects.**

### 1.1 Creation of a new project

To create a new project, click "File -> New Project". As a result, the "New Project" wizard opens. Now, select "Templates -> Visual C++ -> Win32" and make sure "Win32 Console Application" is highlighted in the right side of the window (see Figure 58).
Enter a name for the new project and the new solution and make sure the location for the new project directory is set correctly. Now press "OK" to continue.

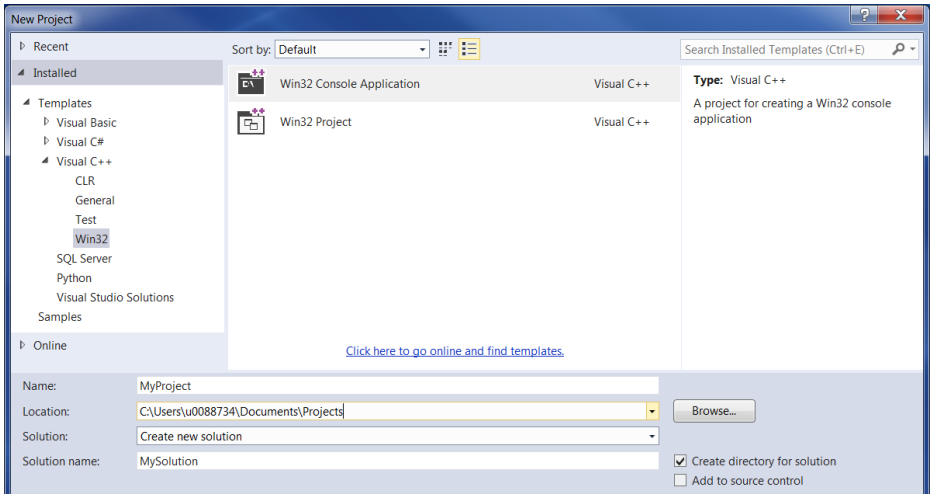



**Figure 58: New Project window**

A new window called "Win32 Application Wizard" opens. Click "next" to open the "application settings" window. Make sure to unselect "Precompiled header" and "Security Development Lifecycle (SDL) checks" and select "Empty project" (see Figure 59). Now click "Finish" to start the creation of the new project.

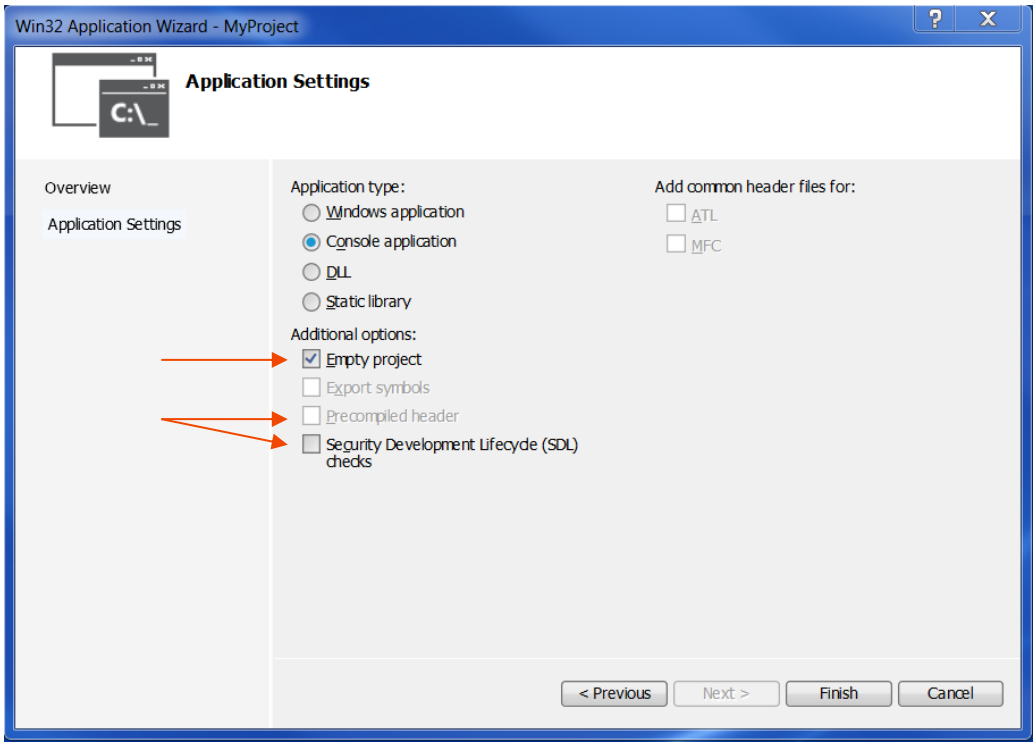



**Figure 59: application settings window**

## 1.2 Creation of a new source file

To create a new source file, right click "source files" in the "solution explorer" and choose "add -> new item" (see Figure 60).

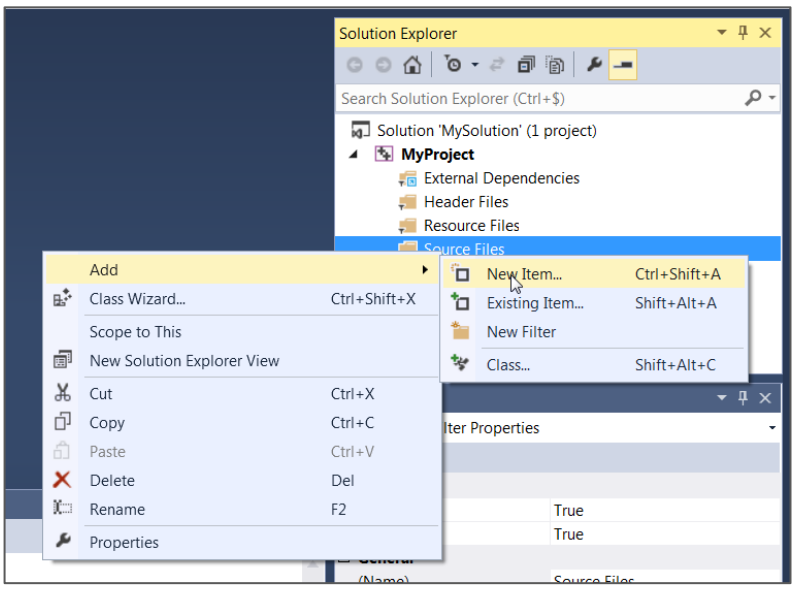



**Figure 60: add new item**

The "Add New Item" window opens. Now, select "Visual C++ -> Code" and make sure "C++ File(.cpp)" is highlighted in the right side of the window (see Figure 61). Since we will write C code and not C++ code, the name of the source file must end in ".c" instead of the suggested ".cpp". Click "Add" to create the new source file.

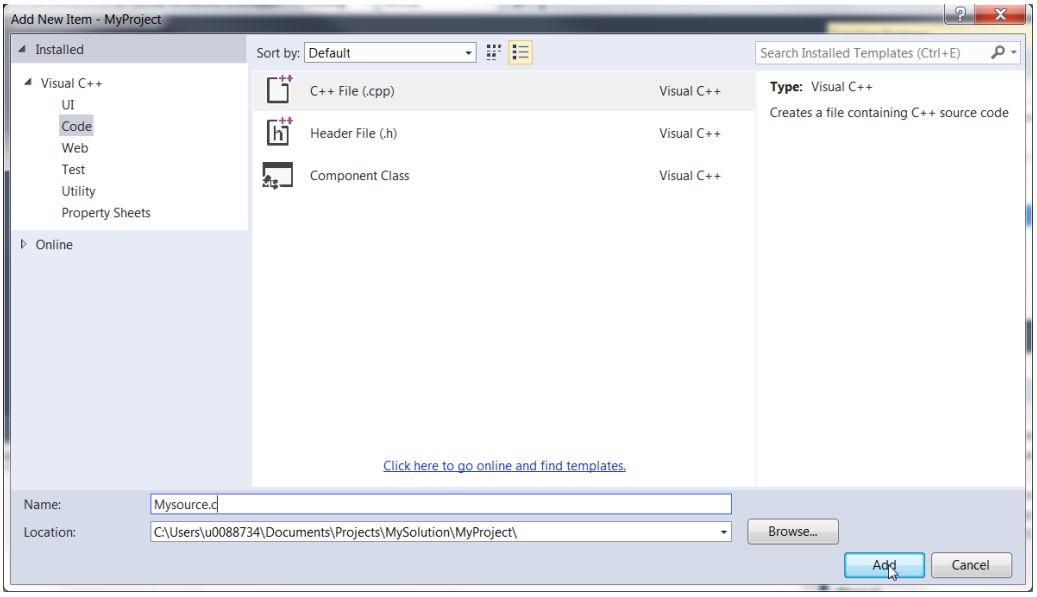



**Figure 61: Add New Item window**

## 1.3 Compile and run a program

Once the source code is written, the program can be compiled and run by clicking the green button "Local Windows Debugger".

By default, the program is run in debug mode showing all screen output in a console window that will disappear at the end of the program. To keep the window active, add a breakpoint before the statement "`return 0;`" in the main function or precede that statement with "`getchar();`" to make sure the program waits for a user input.

If you want to run the program without debugging, press CTRL-F5 instead.

## 2. ASCII table

| Dec | Hex | Oct | Bin | Char | Dec | Hex | Oct | Bin | Char | Dec | Hex | Oct | Bin | Char | Dec | Hex | Oct | Bin | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0x00 | 000 | 0000000 | NUL | 32 | 0x20 | 040 | 0100000 | space | 64 | 0x40 | 100 | 1000000 | @ | 96 | 0x60 | 140 | 1100000 | ` |
| 1 | 0x01 | 001 | 0000001 | SOH | 33 | 0x21 | 041 | 0100001 | ! | 65 | 0x41 | 101 | 1000001 | A | 97 | 0x61 | 141 | 1100001 | a |
| 2 | 0x02 | 002 | 0000010 | STX | 34 | 0x22 | 042 | 0100010 | " | 66 | 0x42 | 102 | 1000010 | B | 98 | 0x62 | 142 | 1100010 | b |
| 3 | 0x03 | 003 | 0000011 | ETX | 35 | 0x23 | 043 | 0100011 | # | 67 | 0x43 | 103 | 1000011 | C | 99 | 0x63 | 143 | 1100011 | c |
| 4 | 0x04 | 004 | 0000100 | EOT | 36 | 0x24 | 044 | 0100100 | $ | 68 | 0x44 | 104 | 1000100 | D | 100 | 0x64 | 144 | 1100100 | d |
| 5 | 0x05 | 005 | 0000101 | ENQ | 37 | 0x25 | 045 | 0100101 | % | 69 | 0x45 | 105 | 1000101 | E | 101 | 0x65 | 145 | 1100101 | e |
| 6 | 0x06 | 006 | 0000110 | ACK | 38 | 0x26 | 046 | 0100110 | & | 70 | 0x46 | 106 | 1000110 | F | 102 | 0x66 | 146 | 1100110 | f |
| 7 | 0x07 | 007 | 0000111 | BEL | 39 | 0x27 | 047 | 0100111 | ' | 71 | 0x47 | 107 | 1000111 | G | 103 | 0x67 | 147 | 1100111 | g |
| 8 | 0x08 | 010 | 0001000 | BS | 40 | 0x28 | 050 | 0101000 | ( | 72 | 0x48 | 110 | 1001000 | H | 104 | 0x68 | 150 | 1101000 | h |
| 9 | 0x09 | 011 | 0001001 | TAB | 41 | 0x29 | 051 | 0101001 | ) | 73 | 0x49 | 111 | 1001001 | I | 105 | 0x69 | 151 | 1101001 | i |
| 10 | 0x0A | 012 | 0001010 | LF | 42 | 0x2A | 052 | 0101010 | * | 74 | 0x4A | 112 | 1001010 | J | 106 | 0x6A | 152 | 1101010 | j |
| 11 | 0x0B | 013 | 0001011 | VT | 43 | 0x2B | 053 | 0101011 | + | 75 | 0x4B | 113 | 1001011 | K | 107 | 0x6B | 153 | 1101011 | k |
| 12 | 0x0C | 014 | 0001100 | FF | 44 | 0x2C | 054 | 0101100 | , | 76 | 0x4C | 114 | 1001100 | L | 108 | 0x6C | 154 | 1101100 | l |
| 13 | 0x0D | 015 | 0001101 | CR | 45 | 0x2D | 055 | 0101101 | - | 77 | 0x4D | 115 | 1001101 | M | 109 | 0x6D | 155 | 1101101 | m |
| 14 | 0x0E | 016 | 0001110 | SO | 46 | 0x2E | 056 | 0101110 | . | 78 | 0x4E | 116 | 1001110 | N | 110 | 0x6E | 156 | 1101110 | n |
| 15 | 0x0F | 017 | 0001111 | SI | 47 | 0x2F | 057 | 0101111 | / | 79 | 0x4F | 117 | 1001111 | O | 111 | 0x6F | 157 | 1101111 | o |
| 16 | 0x10 | 020 | 0010000 | DLE | 48 | 0x30 | 060 | 0110000 | 0 | 80 | 0x50 | 120 | 1010000 | P | 112 | 0x70 | 160 | 1110000 | p |
| 17 | 0x11 | 021 | 0010001 | DC1 | 49 | 0x31 | 061 | 0110001 | 1 | 81 | 0x51 | 121 | 1010001 | Q | 113 | 0x71 | 161 | 1110001 | q |
| 18 | 0x12 | 022 | 0010010 | DC2 | 50 | 0x32 | 062 | 0110010 | 2 | 82 | 0x52 | 122 | 1010010 | R | 114 | 0x72 | 162 | 1110010 | r |
| 19 | 0x13 | 023 | 0010011 | DC3 | 51 | 0x33 | 063 | 0110011 | 3 | 83 | 0x53 | 123 | 1010011 | S | 115 | 0x73 | 163 | 1110011 | s |
| 20 | 0x14 | 024 | 0010100 | DC4 | 52 | 0x34 | 064 | 0110100 | 4 | 84 | 0x54 | 124 | 1010100 | T | 116 | 0x74 | 164 | 1110100 | t |
| 21 | 0x15 | 025 | 0010101 | NAK | 53 | 0x35 | 065 | 0110101 | 5 | 85 | 0x55 | 125 | 1010101 | U | 117 | 0x75 | 165 | 1110101 | u |
| 22 | 0x16 | 026 | 0010110 | SYN | 54 | 0x36 | 066 | 0110110 | 6 | 86 | 0x56 | 126 | 1010110 | V | 118 | 0x76 | 166 | 1110110 | v |
| 23 | 0x17 | 027 | 0010111 | ETB | 55 | 0x37 | 067 | 0110111 | 7 | 87 | 0x57 | 127 | 1010111 | W | 119 | 0x77 | 167 | 1110111 | w |
| 24 | 0x18 | 030 | 0011000 | CAN | 56 | 0x38 | 070 | 0111000 | 8 | 88 | 0x58 | 130 | 1011000 | X | 120 | 0x78 | 170 | 1111000 | x |
| 25 | 0x19 | 031 | 0011001 | EM | 57 | 0x39 | 071 | 0111001 | 9 | 89 | 0x59 | 131 | 1011001 | Y | 121 | 0x79 | 171 | 1111001 | y |
| 26 | 0x1A | 032 | 0011010 | SUB | 58 | 0x3A | 072 | 0111010 | : | 90 | 0x5A | 132 | 1011010 | Z | 122 | 0x7A | 172 | 1111010 | z |
| 27 | 0x1B | 033 | 0011011 | ESC | 59 | 0x3B | 073 | 0111011 | ; | 91 | 0x5B | 133 | 1011011 | [ | 123 | 0x7B | 173 | 1111011 | { |
| 28 | 0x1C | 034 | 0011100 | FS | 60 | 0x3C | 074 | 0111100 | < | 92 | 0x5C | 134 | 1011100 | \ | 124 | 0x7C | 174 | 1111100 | \| |
| 29 | 0x1D | 035 | 0011101 | GS | 61 | 0x3D | 075 | 0111101 | = | 93 | 0x5D | 135 | 1011101 | ] | 125 | 0x7D | 175 | 1111101 | } |
| 30 | 0x1E | 036 | 0011110 | RS | 62 | 0x3E | 076 | 0111110 | > | 94 | 0x5E | 136 | 1011110 | ^ | 126 | 0x7E | 176 | 1111110 | ~ |
| 31 | 0x1F | 037 | 0011111 | US | 63 | 0x3F | 077 | 0111111 | ? | 95 | 0x5F | 137 | 1011111 | _ | 127 | 0x7F | 177 | 1111111 | DEL |