

Introduction to Computing

EP Course
WS 2009/10 <ZP>

Core Lecture Notes

© Prof. Dr. B. Bartning, Emden
University of Oldenburg
<http://www.bartning.org>

Date: 30th September 2009

Contents

0	Introductory remarks	4
0.1	Legend, language remarks, picture symbols	4
0.2	Lecture notes	4
0.3	Advices for applying older C++ compilers	5
I	Introduction to Computing (course)	6
1	Computer systems (short overview)	6
1.0	Overview	6
1.1	Basic terms	6
1.2	Computer structure, hardware	7
1.3	Software, operating systems	8
1.4	Numbers, characters	12
2	Algorithm structures, operators (language independent reflection)	17
2.0	Overview	17
2.1	Introduction to algorithms	17
2.2	Sequence	18
2.3	Selection	21
2.4	Iteration	25
2.5	Nesting the three algorithmic structures	27
2.6	Operators, logical combinations	30
3	First steps with C++	33
3.0	Overview	33
3.1	The first programs	33
3.2	Simple data types, strings, operators	34
3.3	Expressions, side effects	36

4	Algorithm structures in C++	38
4.0	Overview	38
4.1	Sequence, scope within blocks	38
4.2	Boolean expressions, data type <code>bool</code>	38
4.3	Selection	39
4.4	Iteration	41
4.5	The <i>condition</i> in control structures, scopes with more recent compilers	42
4.6	Example	43
4.7	Recommendations for naming and layout, symbolic constants	44
5	Input and output, array and string type, supplements	47
5.0	Overview	47
5.1	Overview of operators	47
5.2	Standard streams, error handling	50
5.3	Handling text files	55
5.4	Arrays	58
5.5	Strings (C strings)	59
5.6	Overview of types	60
5.7	Preprocessor	62
6	Creating construct units, problem of distinction between hidden and accessible (language-independent reflection)	65
6.0	Overview	65
6.1	Design of systems: construct units and hiding principle	65
6.2	Procedural programming	67
6.3	Storage classes, modular programming	71
6.4	Object oriented programming	73
7	Procedural programming	75
7.0	Overview	75
7.1	Function definition and function call, scope within blocks	75
7.2	Function declaration	79
7.3	Reference type	80
7.4	Parameter passing modes value and reference	81
7.5	Global and local names	84
7.6	Overload of function names, default arguments	85
7.7	Recursive functions	87
7.8	Function guidelines	88
8	Storage classes, modular programming	90
8.0	Overview	90
8.1	Static and automatic storages class	90
8.2	Modular programming: splitting into several translation units	92

9 Object oriented programming: encapsulation of data and functions with access control	99
9.0 Overview	99
9.1 Classes and objects	99
9.2 Constructor and destructor	101
9.3 Examples	102

II Supplements, not included in the course (text see internet version of the lecture notes)

10 Some C++ supplements, pointer type, free store

10.0 Overview	
10.1 Symbolic constants, macros and inline functions	
10.2 Data type pointer, type interpretation, array and pointer	
10.3 Free store	

11 Object orientation: supplements, inheritance, polymorphism, static class members

11.0 Overview	
11.1 Classes and objects: supplements	
11.2 Operator overloading	
11.3 Single inheritance	
11.4 Polymorphism	
11.5 Static class members	

12 Operators, types, supplements to pointers, binary files

12.0 Overview	
12.1 Operators	
12.2 Enumeration type, type conversion	
12.3 Pointer arithmetic	
12.4 Pointer as function parameters	
12.5 Applications	
12.6 Dealing with binary files	
12.7 Multidimensional arrays and pointers, command line parameters	
12.8 Generic pointer, C library functions	

0 Introductory remarks

0.1 Legend, language remarks, picture symbols

Explanations

<code>xyz</code>	terminal “xyz” (use these characters or words verbatim)		
<code>abc</code>	meta symbol: replace “abc” by definition of <code>abc</code>		
\doteq	definition symbol (if applicable also within an alternative) —in contrast terminal equal sign: =		
	meta symbol for alternative—in contrast terminal vertical bar:		
{ }	meta bracket symbols—in contrast terminal square brackets: []		
<code>abc_{0..n}</code>	zero or more times “abc”, sequence of an arbitrary number of “abc” (also zero)		
<code>abc_{1..n}</code>	one or more times “abc”, sequence with at least one “abc”		
<code>abc_{opt}</code>	optional “abc” (same meaning as: <code>abc_{0..1}</code>)		
$\left[\begin{array}{l} abc \\ def \end{array} \right]$	equivalent to: $\{ abc \mid def \}$	[NC]	definition not complete
Op2 ^{Op2}	reference to operator (hierarchy level)	[Forb]	forbidden, not allowed (in this course)
¹⁰	reference to syntax element	<text>	special term in German
		[...]	explanation (not belonging to syntax)

Language notes:

(*nothing*) in C and C++

`C++` only in C++

`C` only in C


`C/C++` in C; also possible and usual in C++ (mostly used as contrast to `C++`)


`C (C++)` in C; in C++ possible, too, but not recommended: in C++ better constructs available


`C++(new)` C++, new, perhaps not yet implemented in current compilers


`C++(old)` C++, old, perhaps still implemented in current compilers


Pictograms:


 *Warning: remarks about risks; please, pay attention, take it seriously!*

 *Recommendation: mostly a suggestion, preferable to other possibilities because they are less usual or because they are riskier—at least according to the current state of knowledge*

 *Note: additional note to the text*

 *Remark: explanation in larger context or additional comments not relevant for the examination*


 *Forward reference: reference to later contents of the course*

 Repetition, backward reference

 New—in contrast to repetition

 *Summary*

 Overview

 Example

 Proof

0.2 Lecture notes

Copyright: Please, keep in mind that all lecture notes are **for personal study only**. Further circulation or use is prohibited.

The notes “Language C++” are often cited here with the form (C++/ch. *chapter.point*). They are available as a PDF file on the server. Additionally, you will receive one copy. You may use these notes—without self-made supplements—during the examination(s), however keep in mind that no lecture notes are allowed.

For citing course points, the form (*chapter.point*) is used.

The lecture notes are related to so-called run identifications; a student belongs to the same identification if he or she does not make any pauses in the study. The run identification belonging to these lecture notes is <ZP> (cp. head line).

Examples:

```
run identification <Y>: begin WS 2007/08, semester 2 in SS 2008
run identification <Z>: begin WS 2008/09
run identification <ZP>: begin WS 2009/10
```

If applicable, also books are cited:

K&R2/ Kernighan/Ritchie, The C Programming Language, 2nd edition, Prentice Hall 1988;
German: Programmieren in C, 2. Auflage, Hanser 1990

Str2/ Stroustrup, The C++ Programming Language, 2nd edition, Addison-Wesley 1991;
German: Die C++ Programmiersprache, 2. Auflage, Addison-Wesley 1992 and later
(german edition: pay attention, many errors!)

Str3/ Stroustrup, The C++ Programming Language, 3rd edition, Addison-Wesley 1997;
German: Die C++ Programmiersprache, 3. Auflage, Addison-Wesley 1998

EffCpp/ Meyers, Effective C++: 50 Specific Ways ..., 2nd edition, Addison-Wesley 1998;
German: Effektiv C++ programmieren, 3. Auflage, Addison-Wesley 1998

MEffCpp/ Meyers, More Effective C++: 35 Specific Ways ..., Addison-Wesley 1997;
German: Mehr Effektiv C++ programmieren, Addison-Wesley 1997

D&E/ Stroustrup, Design and Evolution of C++, AT&T Bell Lab. 1994;
German: Design und Entwicklung von C++, Addison-Wesley 1994

ARM/ Ellis/Stroustrup, The Annotated C++ Reference, Addison-Wesley 1990

0.3 Advices for applying older C++ compilers

This course and the exercises are intended to be used with new compilers.

If an **older C++ compiler** is available:

- Perhaps, the compiler does not know the data type `bool`; in this case, please include the following lines:

```
typedef int bool;
const bool true=1, false=0;
```

- If you define variables in the initialization part of a *for-statement*: enclose the complete *for-statement* in a block, if applicable, see (4.51c).
- Presumably, it does not yet know the include lines in the cited form. In this case, please replace the lines by lines like the following example lines:

```
// C++ include files
#include <iostream.h>
// instead of: #include <iostream> (i. e. add ".h")

// C include files:
#include <string.h>
// instead of: #include <cstring> (i. e. omit prefix "c",
// and add ".h")
```

Additionally, you have to omit the following line, too:

```
using namespace std;
```

Some detail of this are indicated in (8.23c).

Very important: In any case, you should not mix these two notations of the include lines in a project!

Part I

Introduction to Computing (course)

1 Computer systems (short overview)

1.0 Overview

In this chapter you learn about hardware and software components of a computer and their interaction. It is a short collection of the most important items. The subject will hardly be explained; it is a short summary of what is important for the further parts of the course. For students not knowing very much about computers, this overview (even together with the explanations in the lectures) may be too short, and therefore it may not be grasped. Please, in this case read introducing computer science books about these items (see library!).

1.1 Basic terms

(1.10) Ovw The basic terms (which will be mentioned again and again in the lectures) have to be learned: hardware, software, program, analogue and digital, character, bit, byte and the front characters K, M, G, T.

(1.11) Some terms—partly as *provisional* explanations:

- (a) **Hardware** ⟨Hardware⟩: physically-technically realized parts of a computer system.
- (b) **Software** ⟨Software⟩: all programs.
Program ⟨Programm⟩: collection of instructions to solve a special task, written so that a computer can understand it directly/indirectly—see also (2.11).
- (c) **Data** ⟨Daten⟩: information (denotations about facts or events/processes) based upon known agreements in a form processable by machine.

(1.12) Representing and transporting information is mostly done in form of physical quantity.

Ex Electrical voltage, distance (movement in space), brightness, magnetic field, temperature, and many more.

If the physical quantity takes on only certain, unambiguously distinguishable values, we call it **digital** information (or representation). If it may take on—within a certain range—each arbitrary value, it is called **analogue** information (or representation).

Ex Analogue: clock with hands (without step motion), physical oscillation, continuous movement.
Digital: digital clock, switch on/off, movement by step motor.

Digital/analogue computer: computer that internally links the data digitally/analogue.

(1.13) **Character** ⟨Zeichen⟩: element out of a finite set for representing information (for digital information representation). Codes see (1.45).

In data processing, the following are usual:

- digits (0..9),
- letters (A..Z, a..z, additional also national special letters, in German e. g. ÄÖÜäöüß),
- special characters (e. g. + - . , ; * / @ \$ and others).

(1.14) **Bit** (from ‘binary digit’): smallest information unit, can represent only two states (e. g. 0/1, yes/no, true/false, current/no current, bright/dark).

Byte = 8 bits (nowadays). Per byte there are $2^8 = 256$ distinct bit patterns. Today a character (1.13) is often represented in one byte, newer codes (e. g. Unicode) have 2 bytes.

1 k = 10^3 (e. g. 1 km, 1 kg) – but:

1 K = $2^{10} = 1024 \gtrsim 1000$

Ex 1 Kbyte = 1024 bytes
1 Kbit = 1024 bits

Order of magnitude: one type writer page has about 2000–3000 characters, i. e. 2–3 Kbytes

1 Mbyte (Mega) = 2^{20} bytes = $(1024)^2$ bytes = 1 048 576 bytes \approx 1 million bytes

1 Gbyte (Giga) = 2^{30} bytes = $(1024)^3$ bytes = 1 073 741 824 bytes \approx 1 billion bytes (American billion)—or 1 thousand million bytes (UK).

1 TByte (Tera) = 2^{40} bytes = $(1024)^4$ bytes = 1 099 511 627 776 bytes \approx 1 trillion bytes (American trillion)

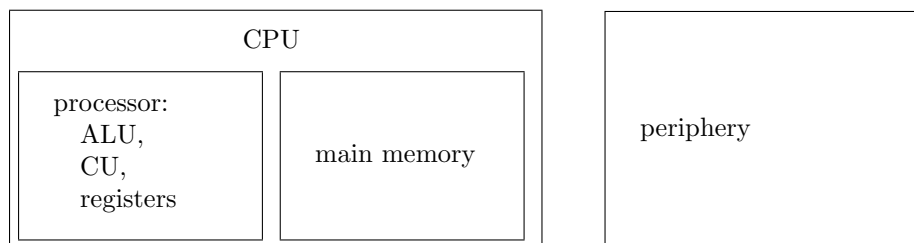
⚠ In physics, the meaning of each prefix is an exact power of ten (like above 1 k), e. g. 1 MHz = 1 000 000 Hz, 1 GHz = 1 000 000 000 Hz.

1.2 Computer structure, hardware

(1.20) Ov At first, the basic structure of a today’s computer (with so-called von Neumann architecture) is presented, i. e. mainly the hardware parts (1.21). Point (1.22) lists the most important periphery devices, (1.23) presents terms for characterizing several aspects of memory.

(1.21)

(a) Principle computer structure:



ALU ‘arithmetic logical unit’ (Rechenwerk): arithmetic operations, comparisons, address calculations.

CU ‘control unit’ (Leitwerk, Steuerwerk): controls the instruction processing, decodes the stored instructions.

Registers (Register): few, very fast memory spaces, number e. g. 16.

Processor (Prozessor): Combination of ALU, CU, registers.

Main memory (Hauptspeicher), RAM (see below): cabinet with numbered compartments (‘addresses’). Nowadays, each address mostly contains 1 byte.

CPU ‘central processing unit’ (Zentraleinheit): Combination of processor and main memory.

⚠ With microprocessors, there is a different meaning: CPU = processor without main memory.

Periphery, periphery devices, input/output devices (Ein-/Ausgabegeräte): everything outside the CPU.

(b) Order of magnitude for transfer durations:

- ALU–register: 1–2 ns,
- ALU–main memory: about 10 ns,
- ALU–hard disk: about 10 ms,
- ALU–magnetic tape: up to sec.

Attention: factor 10^6 between 10 ns and 10 ms! However, the transfer of following bytes from a hard disk is much faster—in contrast to the main memory.

To bridge large speed differences, cache memories are used very often, e. g. disk cache (mostly on disk device), external and internal cache (between main memory and ALU).

(c) The original meaning of RAM and ROM can be misunderstood:

- RAM (from ‘random access memory’) now has the meaning read-write memory in the form of the main memory. It has the access mode direct (synonymous: random),
- ROM (from ‘read-only memory’) has just this self explanatory meaning: memory for reading only, in contrast to the read-write memory. Access mode here also direct (random) usually.

(1.22) Periphery

- (a) External memory, mass storage: all memory devices outside the CPU.
- magnetic tape, streamer (access mode: sequential)
 - storage with magnetic platters, hard disk;
 - organization: several platter surfaces, sector ⟨Sektor⟩, track ⟨Spur⟩, cylinder ⟨Zylinder⟩ (the latter: combination of all tracks of the same number);
 - access mode: half direct (track/cylinder direct, sector indirect)
 - floppy disk ⟨Diskette⟩
 - CD (‘compact disk’)
- ⚠ *VERY IMPORTANT: save all important data, in general daily; when you need to restore data, it is too late for backup!!*
- (b) Input devices: keyboard, mouse, optical character reader, bar code reader, sensor (for process computer), microphone, (graphics) tablet, electronic pen, scanner, CD read device; formerly: paper tape reader, punched card reader.
 ⟨Tastatur, Maus, Klarschriftleser, Balkencodeleser, Prozessfühler (Sensor), Mikrofon, Tablet, Lichtgriffel, Scanner, CD-Lesegerät; Lochstreifenleser, Lochkartenleser.⟩
- (c) Output devices: screen, printer, plotter, loud speaker, CD device (burner); formerly: paper tape punch, card punch.
 ⟨Bildschirm, Drucker, Plotter, Lautsprecher CD-Gerät (Brenner); Lochstreifenstanzer, Lochkartenstanzer.⟩

(1.23) Memory categories

according to ...	categories	e. g.
access (to a special location)	random access (access time independent from location of former access)	main memory
	sequential access (access one part after the other)	streamer (tape)
	mixture	hard disk, floppy disk, also CD
direction of information flow	read only	ROM chip CD ROM
	read and write	main memory, hard disk, streamer
	mixture	WORM, EPROM

1.3 Software, operating systems

- (1.30) Ow This subchapter is an overview of the software of a computer system. You learn about language generations, then the steps from source code to an executable program. After this, several kinds of software are explained, in this the term ‘operating system’ is defined. In (1.34) a hierarchical file system is introduced (DOS/Windows and others, mainly also Unix); it is important that you know how to navigate in such a hierarchy, and how to express paths (absolute and relative). You may get some interesting effects with the aid of the operating system by using redirection, piping, and filter, often easier than via direct programming.
- (1.31) Different language generations
- Note *This generation counting isn’t usual today except the ‘4GL’.*
- (a) **Machine language** ⟨Maschinensprache⟩ (first generation language), can be understood by

the processor directly.

Ex Executable programs.

- (b) **Assembler, low-level language** (Assembler, maschinenorientierte Sprache) (second generation language): for each machine instruction there is one assembler instruction, very dependent on the processor, for humans more readable than machine code.

Translation program for source code written in this language: **assembler** (Assembler, Assemblierer). Double meaning of this word: language, and translation program; without ambiguity: assembly language, and assembly program.

- (c) **High-level (programming) language, problem-oriented language** (höhere Programmiersprache, Hochsprache; problemorientierte Sprache) (third generation language): is more oriented to the problem which has to be solved than to the executing processor.

Ex Fortran, Cobol, Algol, Basic, Pascal, C, C++.

Translation program for source code in this language: **compiler** (Compiler, Kompilierer).

- (d) Language of the forth generation (‘4GL’): language which informs the processing unit what result the user wants to have, but not how to get this or in which way to do this.

Ex Some data base query languages.

(1.32) Steps from source text until program execution:

- Write the source code in a **source language** (Quellsprache) (assembler, high-level language, 4GL) with the aid of an editor program: clearly readable text without formatting—except a good layout with line breaks and indentations.
- Translate this source program with a **compiler** or assembler; result: **object program** (partly in machine language, additionally tables with unresolved [i. e. needed] and with offered references).
- Link the object program(s) with a **linker** (Binder, Linker) with the aid of libraries, fulfill the object programs’ unresolved references, create the **executable program** (ausführbares Programm) on the hard disk.
- **Load** (laden) the executable program into the main memory and initiate the **execution** (with the so-called loader).

Within an Integrated Development Environment (‘IDE’), you hardly notice these single steps.

Another procedure is possible in some programming languages with an **interpreter** (Interpreter): the source code (high-level language) is transformed directly into machine code and passed over for execution, after this the generated machine code is discarded.

Advantage: immediate test of the source code. Disadvantage: inconvenient performance. Essential disadvantage: leads into programming without thinking (Basic!).

(1.33)

- (a) Depending on the purpose, you may distinguish several kinds of software:
- processing programs:
 - application programs (salary computing, NC program, test processing, data base program),
 - compiler,
 - utilities (Dienstprogramme) (sort program, test program, data transfer program);
 - system programs (= operating system).
- (b) **Operating system (OS)** (Betriebssystem): the whole of the system programs; in conjunction with the hardware, the OS provides the functional structure and operating mode of the system.

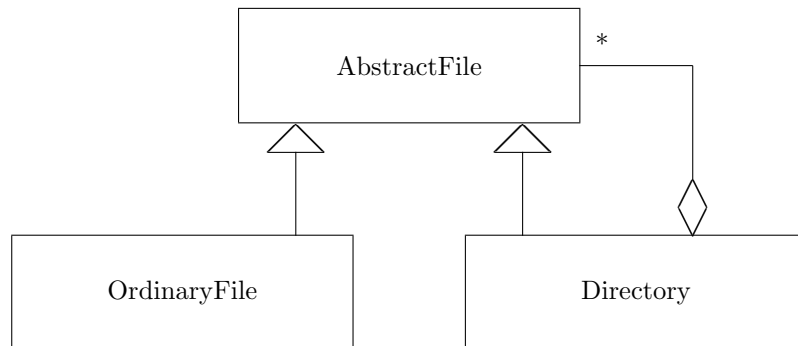
(1.34) Hierarchical file system (DOS, Windows, also Unix in some respects)

- (a) **File** (Datei): collection of information belonging together.

Directory (Verzeichnis, Ordner (folder)): container for files and directories.

A directory may contain directories and/or files, a file cannot contain any file or directory.

Graphical representation (UML, see (2.14b↑↑)):



Explanation:

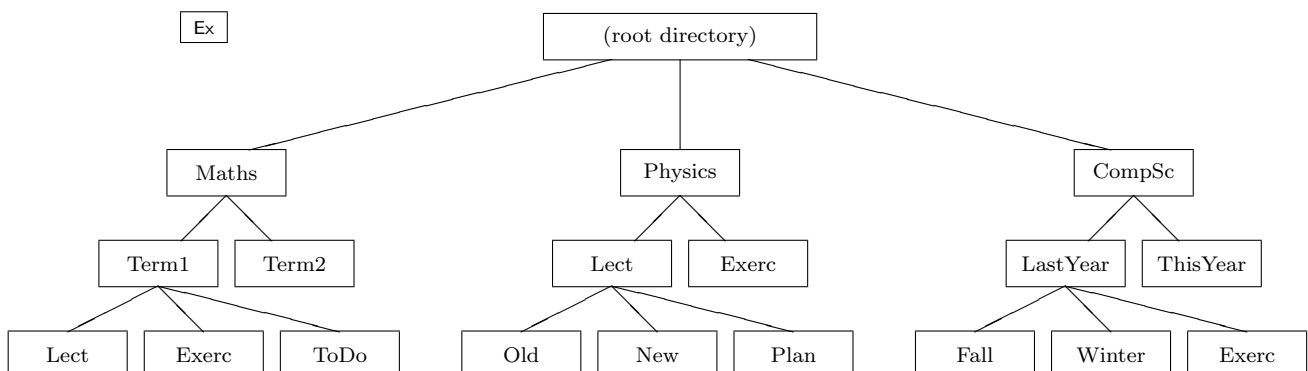
OrdinaryFile (normal file) and Directory have many common features; that what is common is summarized in AbstractFile. The kind of relationship (arrow with empty triangle) is a specialization: OrdinaryFile and Directory are each a special kind of AbstractFile. The denotation 'abstract' refers to the terminology in object orienting; this means that AbstractFile itself cannot exist in reality, but always only in the specialized form OrdinaryFile or Directory.

Additionally, there is a whole-part- or contains-relationship (line with a rhombus) between Directory and AbstractFile (this one, again, as OrdinaryFile or as Directory); the asterisk indicates that this relationship from Directory to AbstractFile may exist to any number (0..n) of these elements.

↑↑ The kind of representation (UML) and the explanation based on it are not relevant for this course, but the meaning (ordinary files, directories, recursive nesting possibilities) is important, nevertheless. In Unix, there are further kinds of files as specialization, e. g. the 'special file' (device file). The above figure is correct for Unix only if there are no additional links. Details about the specialization relationship (object orienting: inheritance) see also (11.31).

(b) Structure of a **directory hierarchy** ('tree')

Root directory (Stammverzeichnis, Wurzelverzeichnis): directory which is not contained in another directory.



Path (Pfad): route from a directory (point) to another directory (point) within a directory tree.

An operating system normally offers to settle on a directory as the default directory (Standardverzeichnis) (and to alter it). Thus, there are two kinds of paths:

- **relative path**: path beginning at the default directory,
- **absolute path**: path beginning at the root directory—independent of the default directory.

(c) DOS/Windows and Unix

Separator (separating character) between path parts: DOS/Windows '\', Unix '/'.

Distinction of the two different kinds of paths:

- heading separator means absolute path,

device shall be regarded as the standard input or output, is not settled by the application program, but by the operating system.

- (b) In DOS/Windows and in Unix, you may alter the mapping on the command line level (**redirection** <Umleitung>):

$$\text{command} \quad >\text{outputfile}_{opt} \quad <\text{inputfile}_{opt}$$

Instead of the usual standard output, the output is directed into *outputfile*, instead of the usual standard input, the input is directed from *inputfile*. Additionally, there is the redirection symbol '>>', it means (with files) that the actual content is not deleted, but the new output is appended.

- (c) **Piping** <Datenübergabe>:

Building a command pipe (a chain of sequential commands) is possible on the command line level (DOS/Windows and Unix). Here the standard output of the first command is the standard input of the second one. The piping symbol is '|', e. g.

$$\text{command1} \mid \text{command2}$$

In DOS/Windows such piping is rarely used, in Unix it is applied very often.

- (d) A **filter** <Filter> is a program generating a (changed) standard data output from a standard data input. It is used very often in pipings.

Ex Sort program **sort**, pagewise output on screen **more**.

- (1.36) **Batch file** <Stapelverarbeitungdatei>, Unix: script file <Skriptdatei>: a text file whose lines consist of operating system commands. This file may be executed; the contained commands are processed as if they were inputted directly on the command line level.

1.4 Numbers, characters

- (1.40) Ovw In order to understand and possibly to control some effects of programs, you must know the representation of numbers and characters. The polyadic number system is well known in the form of the decimal system:

Ex $7305 = 7 \times 10^3 + 3 \times 10^2 + 0 \times 10^1 + 5 \times 10^0$

Here, this system is expanded to other bases; especially you will encounter the base 2 very often, but also base 16, more seldom base 8.

To understand the number range of signed types, in (1.42) the representation of negative numbers is explained. Here the two's complement representation is very important, particularly since very many processors and C++ compilers use this representation.

To represent so-called 'real numbers' (1.43), you have to distinguish between fixed point and floating point notation, i. e. as an internal representation in the computer, but also as an external notation for input and output—independent of the internal representation.

You may run into problems searching for errors in a program, if—at run time—there are overflows or underflows (both: exceeding the representable number range) because completely wrong numbers will be produced without noticing it (1.44).

A very short overview of different character codes (1.45) brings this subchapter to an end. You should understand the ASCII structure in principle; you may also remember later on to find here several codes for the special German letters.

- (1.41)

- (a) **Polyadic number system with the base B** ($B \in \mathbb{N} \setminus \{1\}$):

number system, in total with B symbols ('digits') of the values $0, 1, \dots, B-1$ for representation of non-negative integer numbers in the following form:

$$b_n B^n + b_{n-1} B^{n-1} + \dots + b_1 B^1 + b_0 B^0$$

with $b_i \in \{0, 1, \dots, B-1\}$ digits.

The above expression in position notation (with base B known):

$$b b_{n-1} \dots b_1 b_0$$

Also fractional numbers can be represented:

$$\dots + b_0 B^0 + b_{-1} B^{-1} + b_{-2} B^{-2} + \dots$$

$$\dots b_0 . b_{-1} b_{-2} \dots \quad (\text{with the point as decimal symbol})$$

Common: $B = 10$ **decimal system**
 $B = 2$ **binary system**; digits: 0 and 1 (very often in computers)
 $B = 16$ **hexadecimal system**; digits: 0...9, additionally A...F or a...f
 $B = 8$ **octal system**; digits: 0...7

Important for this course: besides decimal system, the binary and the hexadecimal system.

With n positions, in total B^n different numbers may be represented.

Ex $B = 2, n = 8$: 256 different bit patterns in each byte.

- (b) Algorithm always applicable for **conversion** from one arbitrary polyadic number system into another: continuous integer division of the number by the new base (this division can be done in an arbitrary polyadic number system); the remainders—in reverse order—form the converted number in position notation.

Ex $140_{10} = 12012_3$ (the index number indicates the base); here the calculation belonging to it:

$$\begin{array}{r} 140 \div 3 = 46 + 2 \div 3 \\ 46 \div 3 = 15 + 1 \div 3 \\ 15 \div 3 = 5 + 0 \div 3 \\ 5 \div 3 = 1 + 2 \div 3 \\ 1 \div 3 = 0 + 1 \div 3 \end{array} \quad (\text{finish when getting the division result } 0)$$

Conversion into our decimal system can also be done as follows, too: multiply each digit with the position value and sum them. (In general sense, this can be done also in another system than the decimal system.)

Ex $12012_3 = 1 \times 3^4 + 2 \times 3^3 + 0 \times 3^2 + 1 \times 3^1 + 2 \times 3^0 = 140$

Conversion of numbers of base 2 into numbers of base 8 or 16 (similar also vice versa): combination of three or four bits to a digit, beginning with the last digit.

Ex $140_{10} = 10001100_2 = 8C_{16}$ (from 1000 1100) = 214_8 (from 10 001 100)

- (c) For special cases, there are other codes:

- ‘BCD’ (binary coded decimal): each decimal digit is coded binary: you need 4 bits (a tetrad) for each digit. There are six invalid bit patterns, called pseudo tetrads: difficult calculation, but sometimes necessary because here rounding as in decimal system. Application: business calculation.
- When scanning distance or angle differences, the binary system is useless: if the scanner is not straight or if the bit transition is fuzzy, there are error values between the numbers that may vary very strongly. Remedy: codes with only one changing bit for transition between adjacent numbers. Example: Gray code.

(1.42) Negative integral numbers

- (a) Negative integral numbers can be represented in different ways:

- Additional sign bit:
our usual notation in decimal system; for computers inconvenient, in general, but used with floating point numbers for the mantissa (1.43b).
- $(B - 1)$ **complement**, with $B = 2$: **one’s complement** (Einerkomplement)
The number with the same absolute value, but of opposite sign is generated by subtraction of each digit from $B - 1$, i. e. from the highest possible digit—or by subtraction of the number from $B^s - 1$ with s the number of used digits.
With $B = 2$: invert each digit (each bit); highest bit (‘MSB’, most significant bit) 0 means positive number (or zero), highest bit 1 means negative number (or zero).
Inconvenient calculation (formerly with some computers); curiosity: there are two zeros (‘positive zero’ and ‘negative zero’).

- **B complement**, with $B = 2$: **two’s complement** (Zweierkomplement)

The number with the same absolute value, but of opposite sign is generated by subtraction of the number from B^s with s the number of used digits—or by creating the $(B - 1)$ complement and subsequent addition of 1.

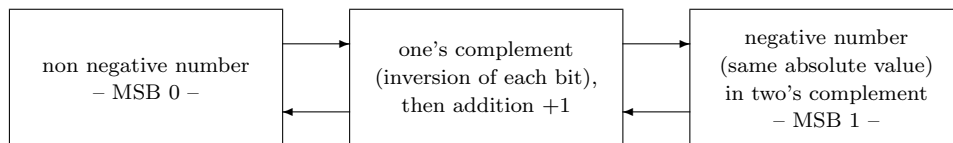
With $B = 2$: invert each digit (each bit), afterwards add 1; highest bit 0 means positive number (or zero), highest bit 1 means negative number. More details and examples see (b)

Nowadays very often used in computers.

- (b) **Two’s complement** (number system with base 2):

Often used acronyms: MSB (most significant bit), in contrast to LSB (least significant bit).

Conversion diagram, cp. (a):



The number range with s digits: $-2^{s-1} \dots 0 \dots (2^{s-1} - 1)$. There are as many non negative numbers (positive and zero) as negative numbers.

Examples for positive und negative numbers in the two’s complement, with the largest positive and the smallest negative number (s is the number of digits):

digit notation	value	Ex $s = 8$ (1 byte)	value
01...11	$2^{s-1} - 1$	01111111	127
		01111110	126
		00000100	4
		00000010	2
00...01	1	00000001	1
00...00	0	00000000	0
11...11	-1	11111111	-1
11...10	-2	11111110	-2
		10000001	-127
10...00	-2^{s-1}	10000000	-128

(1.43) ‘Real’ numbers

- (a) **Fixed point notation** (Festkommadarstellung): a number notation in which each digit has a value fixed only by position, i. e. the decimal point is settled at a pre-determined or agreed position.

Ex Technical-scientific calculations: representation of integral numbers; the (intended) decimal sign is after the last digit (position of lowest value).

Commercial calculations, e. g. calculating with currency values (mostly two decimals).

Disadvantage: numbers with very small or very large absolute value cannot be represented with the same system.

- (b) **Floating point notation**, half-logarithmic notation (Gleitkommadarstellung, Fließkommadarstellung, halblogarithmische Darstellung): representation as product of mantissa m and power to the base B with exponent n ; the number value results in $m \times B^n$. This notation’s name stems from the fact that the value-correct decimal point is not at a pre-determined position, but depends on the exponent.

With known base, you need to store only **mantissa** and **exponent**. In digital computers, both are stored binary (mantissa with extra sign bit, exponent mostly as characteristic calculated from exponent by an additive correction term in order not to be negative).

To achieve as high a relative accuracy as possible, a floating point number must be **normalized**; hereby, the highest digit in the mantissa is unequal zero (unless the mantissa has the value zero).

- (c) ⚠ Because of implicit **rounding errors**, calculations with floating point numbers may lead to completely wrong results if you use an inappropriate algorithm: difference between

‘pure’ and numerical mathematics. Additionally there may be also errors caused by range under- and overflow, see (1.44).


(1.44) Range underflow and overflow

- (a) **Overflow** (Überlauf): the absolute value of the result is too large for the chosen representation; can occur in floating point notation and in fixed point notation.

In practice: according to the kind of operation, further calculations with false bit pattern or run time error; see also (c).

- (b) **Underflow** (Unterlauf): the absolute value of the result is too small for the chosen representation; can occur in floating point notation and in fixed point notation (the latter only if the decimal point is before the least significant digit, i. e. not with integer numbers).

In practice: according to the kind of operation, tacit set to zero, very seldom resulting in a run time error.

- (c)  In C++/C within the unsigned integer range, the overflow—in both directions—is not regarded as an error! When overflow over the maximum value, there is a loop to zero and upwards, when overflow below zero, there is a loop the the highest possible number and downwards.

(1.45) Codes for character representation

- (a) **ASCII** (American standard code for information interchange) is a very often used 7 bit code for character representation.

Structure of the code:

code (hex)	code (dec)	meaning
00...1F	00...31	control character examples: 0A LF line feed 0C FF form feed 0D CR carriage return
30...39 41...5A 61...7A	48...57 65...90 97...122	decimal digits 0...9 upper case letters A...Z lower case letters a...z
rest 20	rest 32	special characters example: SP space

German special letters: not included within ASCII; in the so-called German reference version of ASCII, they displace some special characters, see table (c).

- (b) Better—and today widely used—is the extension of ASCII to 8 bit: additional 128 characters, value (dec.) 128...255. Unfortunately, there are very many different extensions, the most important ones:
- **ANSI character set** (also used by Windows),
 - **IBM extension** of ASCII (used by DOS), e. g. the so-called code table 437 (oldest IBM extension) or code table 850 (now suggested as standard for German).
- (c) The German special letters including the paragraph character (for laws) in the most important codes (codes hexadecimal, in [] decimal):

character	ANSI (Windows)	IBM extended ASCII (DOS)	German reference version of ASCII *)
Ä	C4 [196]	8E [142]	5B [91] [
Ö	D6 [214]	99 [153]	5C [92] \
Ü	DC [220]	9A [154]	5D [93]]
ä	E4 [228]	84 [132]	7B [123] {
ö	F6 [246]	94 [148]	7C [124]
ü	FC [252]	81 [129]	7D [125] }
ß	DF [223]	E1 [225]	7E [126] ~
§	A7 [167]	F5 [245] ¹⁾ bzw. 15 [21] ²⁾	40 [64] @

Remarks: *) after the []: original ASCII character displaced
¹⁾ only code table 850,
²⁾ both code tables 437 and 850—code value however in the range of ASCII control characters(!), see (a)

2 Algorithm structures, operators (language independent reflection)

2.0 Overview

This chapter covers the basic ways how to formulate a computer program. These explanations are independent of a specific programming language; in the following chapters these basic ideas will be applied to the special language C++.

The central term ‘algorithm’ and its parts, the ‘statements’ with their different kinds, are the subject up to subchapter 5. It is important for the students to get to know these basic structures and to be able to use them—language independent. To represent them, one text and two graphic forms are introduced.

You should carefully make the exercises belonging to this chapter (see internet). The aim is to render comprehension for written algorithms and to improve creativity to ‘invent’ algorithms. Surely, you cannot easily learn this creativity; but exercising and trying out, furthermore thinking about example algorithms should entice this, however.

The last subchapter introduces the term ‘operator’, in addition the most important operators that may be applied on logical expressions (‘propositions’).

2.1 Introduction to algorithms

(2.10) Ow At first, the term **algorithm** is introduced which is very basic within computer science (2.11, 2.12). In this course, it is important up to ch. 7. An algorithm consists of **statements**; the three important kinds of statements are shortly rendered (2.13). To represent algorithms, we use one text and two graphic forms (2.14). These terms will be described in more details in the following subchapters.

(2.11)

(a) **Algorithm** (Algorithmus) (in this course): a collection of instructions, which is complete, unambiguous, not inconsistent, executable, (statically) finite, for solving a set of similar problems with a finite number of steps.

Note *In special cases, e. g. when controlling a process, a dynamic finiteness (‘with a finite number of steps’) is not wanted.*

(b) **Program** (in this course): algorithm, expressed in a language a computer can understand directly or indirectly.

(2.12) The execution of an algorithm will be called a process (Prozess). The unit executing (processing) the algorithm is called a **processor**.

Note *This process should not be confused with the (natural or technical) process of process control (e. g. process computer).*

(2.13)

(a) There are two kinds of instructions when writing an algorithm:

- Instructions which are performed directly,
- control instructions or **control structures** (Kontrollstrukturen): these structures express the conditions or frequency, under which one or more instructions shall be performed.

Here you can distinguish between two subordinate kinds:

- Selection: the processing depends on a condition,
- Iteration: the processing will be performed (possibly) many times—depending on a repeatedly checked condition.

Note *The German term ‘Kontroll...(-Struktur)’ is a bad translation of the English ‘control...’, but this German term is usual. Better would be ‘Steuer(ung)...’.*

(b) According to these three kinds of instructions there are **three important base structures** for writing an algorithm:

- sequence (Folge, Sequenz),
- selection (Auswahl, Selektion),
- iteration (Wiederholung, Iteration).

It has been proved that each problem solution which can be expressed as an algorithm can always be written in such a way that only these three algorithm structures are used. In particular, a direct jump statement (‘goto’) can always be avoided.

(2.14) Ways of representing algorithms

- Description with formal English (or German) text, the so-called **pseudo-code**. There is no standard like ANSI. This course uses the pseudo-code to express structures independently from a special computer language.
- Graphical representations:
 - **Program flow chart** (Programmablaufplan, PAP),
 - **Structure chart** or Nassi Shneiderman diagram (Struktogramm, Nassi-Shneiderman-Diagramm).

Note Both graphical representations are standards.

The program flow chart can be created and altered more easily, but with its use it is possible to develop a program which is hardly understandable and maintainable. The structure chart does not have this disadvantage, but it is more difficult to create and alter. For transferring into a program, the structure chart should be chosen, in any case.

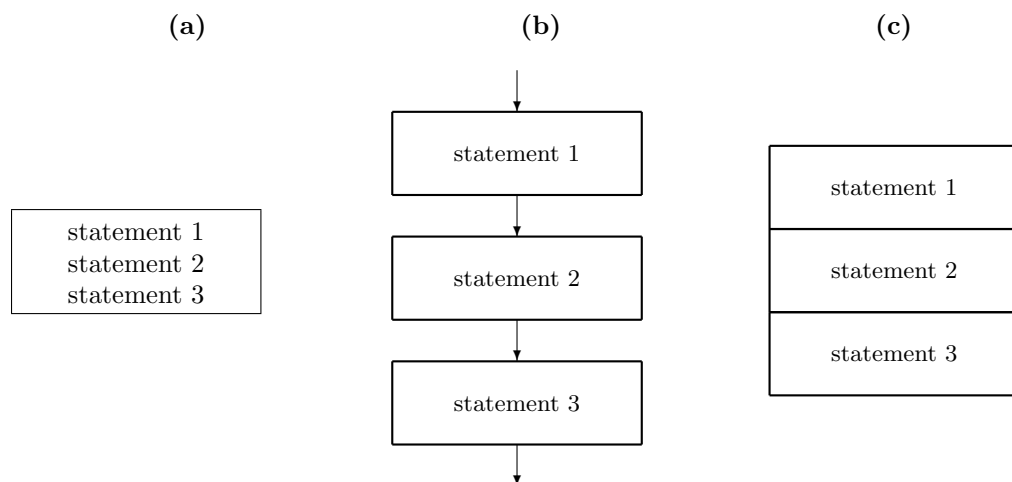
↑↑ For development and representation of complex systems there are different graphical formats. If your development is object oriented, you can use use cases, object diagrams, class diagrams. In the meantime, a graphical language is in common use, it has got a standard (ANSI), the so-called UML (unified modeling language).

2.2 Sequence

(2.20) **Ovw** In a **sequence**, the single parts are processed one after the other without altering the order or missing a part. This very simple base structure for algorithms is explained with the aid of three representations (text and graphic). A full example (2.25) completes this subchapter—please, follow the explanations how to develop the algorithm.

(2.21) The sequence in the three representations:

- pseudo-code, description see (2.22),
- program flow chart, description see (2.23),
- structure chart, description see (2.24).



(2.22) Pseudo-code

- The statements (instructions) (Anweisungen) will be written in a box—according to an

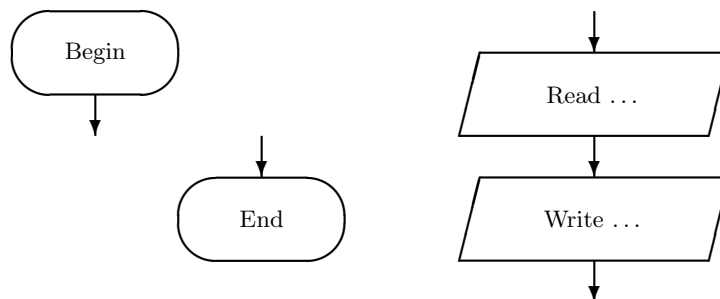
agreement valid for this course. The data memory locations needed for processing (‘boxes’, ‘variables’ with ‘type’) will be mentioned, too, cp. (2.25a).

The name for the algorithm—if existing—will be written in a box above, see (2.25a).

- (b) Rec Important layout recommendation, cp. (2.51) —for this course a must!!—: Each statement shall be written (usually) in one line. If it is too long for a line, it will be continued in the next line, but indented.

(2.23) Program flow chart

- (a) In a program flow chart there are—until now—the following kinds of symbols, see (2.21b) and (b):
- (normal) statement: each in a rectangular box, mostly of constant size,
 - program flow lines (Programmablauflinien): arrows indicating the processing order (for the control flow),
 - begin and end symbols: ovals,
 - special statements for input and output (Eingabe, Ausgabe): parallelograms.
- (b) Symbols for begin/end and for input/output, each drawn with program flow lines:



(2.24) Structure chart

A structure chart always consists of a (large) rectangle with inner (finer) subrectangles. Each inner statement is written in a rectangle, too.

Each structure chart rectangle and each subrectangle will be entered from the upper border line and left through the lower border line.

In the structure chart in this course, there are two additional boxes—not standardized: name of the algorithm and enumeration of the required memory locations (‘variables’), see (2.25c).

(2.25) Ex Calculating the volume of a circular cone

- (a) Pseudo-code:

In the pseudo-code in this course, the so-called ‘keywords’ (words which have a special meaning in the chosen language, in the context) are written with capital letters so that they are emphasized more. Until now we have these keywords:

- (a1) ALGORITHM, BEGIN, END;
- (a2) VARIABLE, TYPE;
- (a3) CONSTANT, VALUE.

Later on, we’ll get some more keywords.

- (1) Preliminary pseudo-code:

ALGORITHM VolCircularCone
BEGIN Use the boxes radius, height, pi, volume Read first number, store it in radius Read next number, store it in height Store 3.14159 in pi Calculate $\frac{1}{3} \times \text{radius} \times \text{radius} \times \text{pi} \times \text{height}$, store the result in volume Write volume END

- (2) Final pseudo-code—often used in this course in this way for representing algorithms:

ALGORITHM VolCircularCone
BEGIN VARIABLE radius, height, pi, volume TYPE floating point number Read radius, height pi ← 3.14159 volume ← radius*radius*pi*height/3 Write volume END

The ‘VARIABLE’ here is in addition connected with a ‘TYPE’. By this, the processor gets to know in which kind and code the storage shall happen—and with it, the size of the memory and the allowed operations. Since the assignment symbols vary with the programming languages, the assignment symbol will be ‘←’ in order to be language independent. The variable names begin with a lower case letter—as usual in English (in contrast to German nouns); the exact recommendation see (4.71b).

- (3) A variation representing this algorithm (a2): Since the number π will always have the same value, we do not assign a value during program execution, but we take a CONSTANT having been set with the correct value before the program runs (‘initialization’):

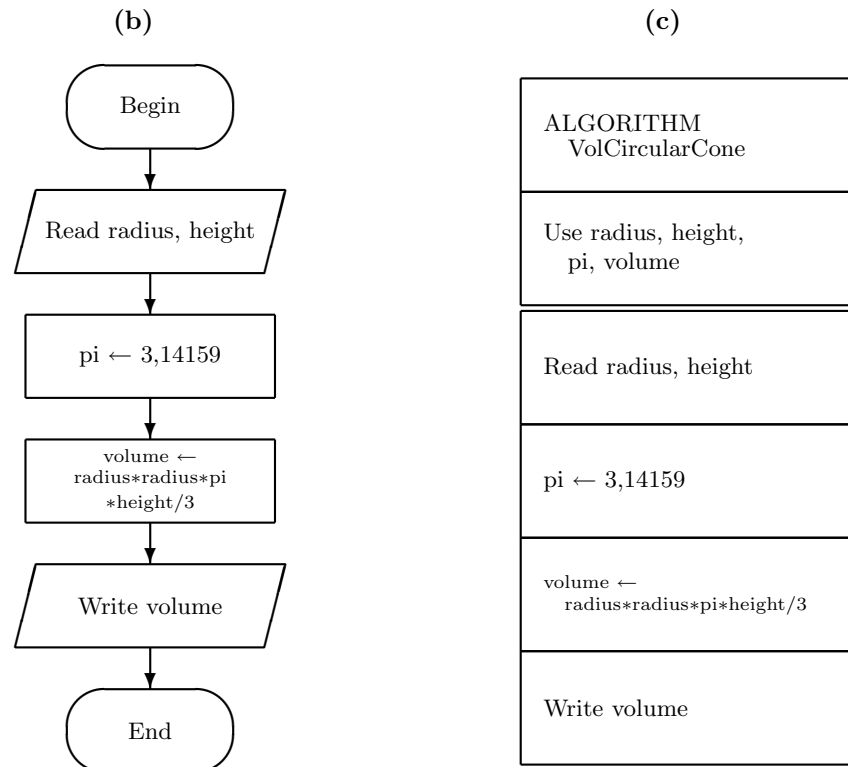
ALGORITHM VolCircularCone
BEGIN VARIABLE radius, height, volume TYPE floating point number // The number pi is introduced as a constant: CONSTANT pi TYPE floating point number VALUE 3.14159 Read radius, height volume ← radius*radius*pi*height/3 Write volume END

We agree—as in the box above—that after the double slash ‘//’ there can be an arbitrary text as a comment; the processor shall ignore all characters from ‘//’ (included) until end of line.

- (4)

<p>How to work with variables (or boxes)—IMPORTANT for grasping the sense of an unknown algorithm:</p> <p>For going through an algorithm, you shall enumerate all variable values, best as a table, one variable per line. If the value is changed, the old value shall be crossed out (but <u>still readable!!</u>), and the new value written behind it in the same line.</p>

- (bc) Program flow chart (b) and structure chart (c) belonging to the representation (a2):



(d) Without comment the algorithm (a3) in C++:

```

// Calculating the volume of a circular cone

#include <iostream>
using namespace std;

int main()
{
    double radius, height, volume;
    const double pi=3.14159;

    cin >> radius >> height;
    volume=radius*radius*pi*height/3;
    cout << "Volume: " << volume << endl;

    return 0;
}
  
```

(2.26) In the early days of computer history, the sequence was the only algorithm structure that could be executed because it was not possible to perform jumps within the program text, neither forwards (2.39) nor backwards (2.45).

2.3 Selection

(2.30) Ov The **selection** (Auswahl, Selektion) is one of the two control structures (2.13a). In a selection, processing a statement depends on a condition (2.31).

There are three kinds of selection statements:

- two branch selection (zweiseitige Auswahl) (IF ... THEN ... ELSE ...),
- one branch selection (einseitige Auswahl) (IF ... THEN ...),
- multiple selection (Mehrfachauswahl), with the subordinate kinds:

- multiple selection as an IF-ELSEIF chain,
- multiple selection with a selector value (CASE DISTINCTION ACCORDING TO ...).

Note The multiple selection with a selector value is not supported by each programming language.

(2.31) The **condition** in a selection or an iteration (ch. 2.4) is a proposition as defined in formal logic.

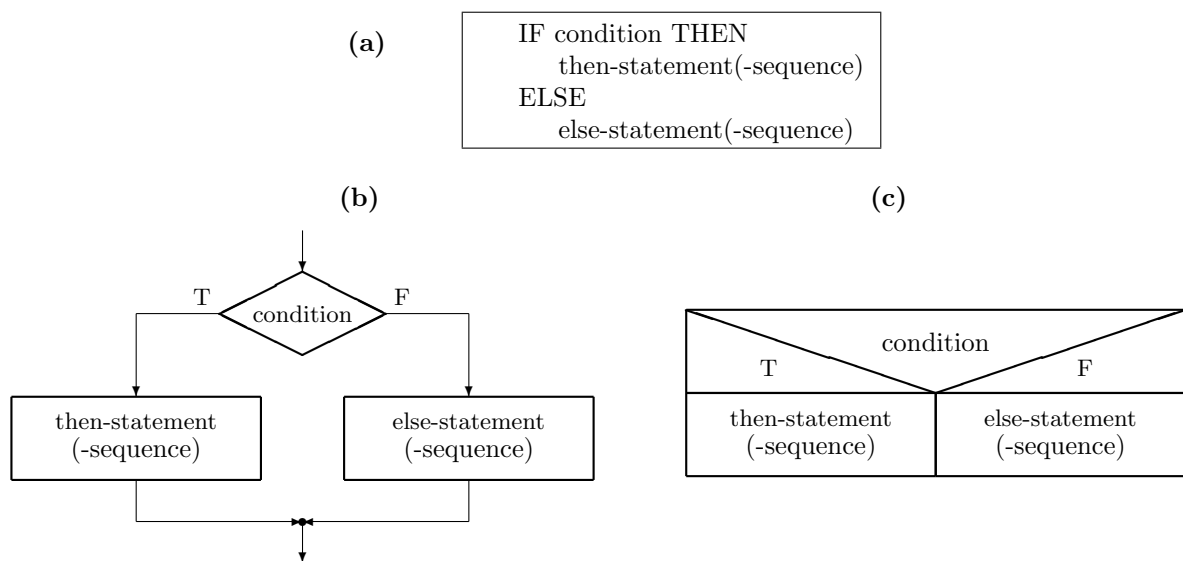
A **proposition** (in formal logic) (Aussage) is a language construct to which the quality TRUE or FALSE can be unambiguously assigned, in principle. The quality TRUE or FALSE is called **truth value** (Wahrheitswert) of the proposition.

Details about propositions, esp. about their operators see (ch. 2.6).

(2.32) In a two branch selection, the condition will be checked, at first. If it is true, the then-statement will be performed, if it is false, the else-statement will be executed.

The two branch selection with the three representations:

- (a) pseudo-code, (b) program flow chart, (c) structure chart:

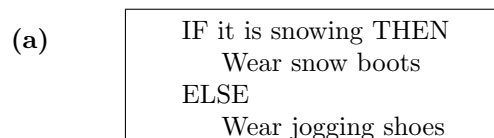


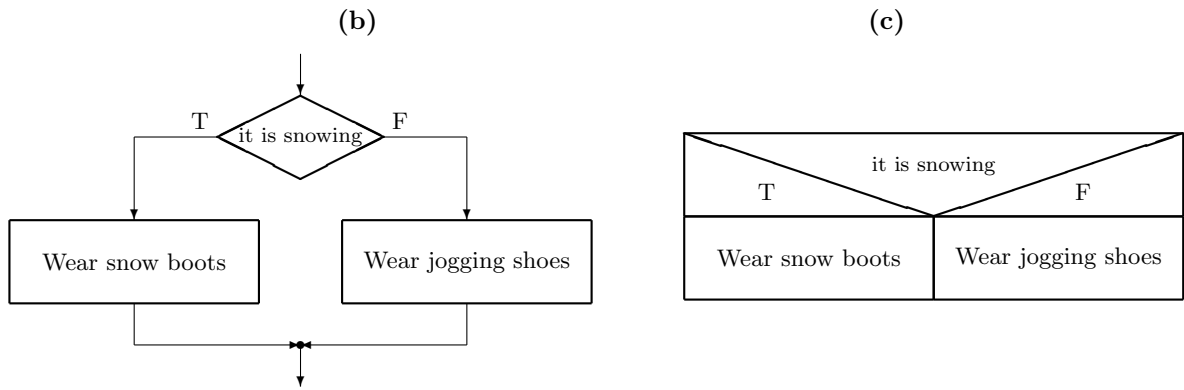
Remarks to (a): The words IF, THEN, ELSE are additional keywords (2.25a).
 Layout (students: a must): Despite the fact that ELSE continues a statement, it shall not be indented—in contrast to (2.22b)—, it shall begin directly under the corresponding IF, see rules (2.51). Explanation for this layout see (2.34↑↑)

Remarks to (b): In the program flow chart, the condition is drawn as a rhombus which has two (or more (2.37b)) exits—here T (true) and F (false).

Remarks to (c): The structure chart will be entered through the upper border line, the triangle will be exited either through the yes-branch or the no-branch.

(2.33) Ex for the two branch selection in the three representations:

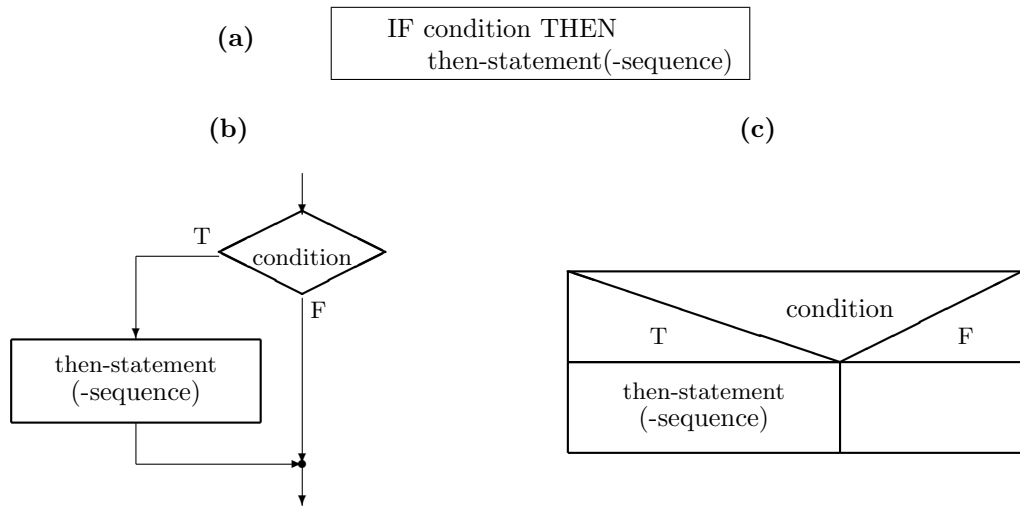




(2.34) The **one branch selection** is a shortening of the two branch selection by omitting the else-branch.

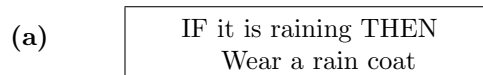
The one branch selection in the three representations:

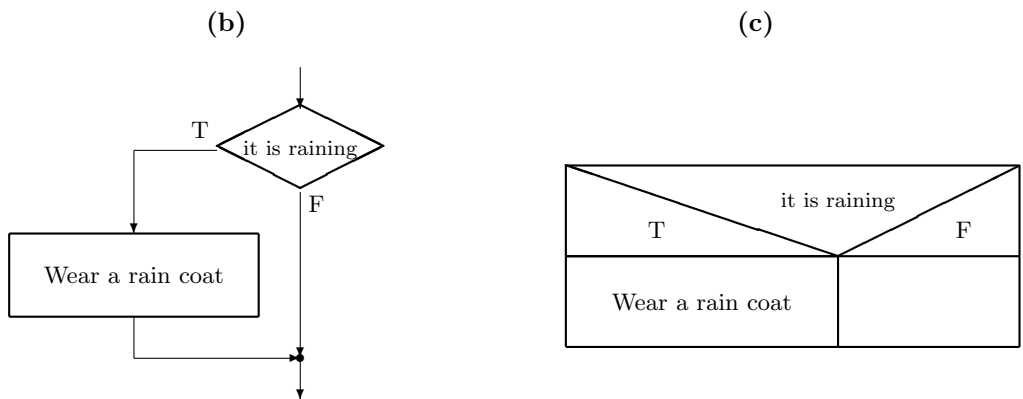
(a) pseudo-code, (b) program flow chart, (c) structure chart:



↑↑ The grammatical ambiguity of the construct 'IF condition THEN statement'—as a complete one branch selection or as the first part of a two branch selection—will be allowed in the pseudo-code because (it is a pity!) it exists in many programming languages, too, also in C++/C. For increasing the clarity—only for the reading human user, not for a compiler—the description in (2.32) holds the recommendation (students: a must) not to indent the distinguishing ELSE to be noticed more easily.

(2.35) Ex for the one branch selection:





(2.36) The multiple selection as an IF-ELSEIF chain will be constructed by nesting two branch selections in several levels, i. e. each subordinate selection is posted into the ELSE branch of the higher level selection. Details about nesting see chapter 2.5.

Rec Layout (for students a must): Since nesting with several levels is not very clear, and since no statement can begin with an ELSE IF, it is recommended to write each ELSE IF and the last ELSE directly under the first (leading) IF, see (2.51), in contrast to (2.22b).

Pseudo-code (left with exact nesting indentation, right with recommended and more readable layout):

```

IF y < 0 THEN
  Write 'y negative'
ELSE
  IF y < 5 THEN
    Write '0 ≤ y < 5'
  ELSE
    IF y < 10 THEN
      Write '5 ≤ y < 10'
    ELSE
      IF y < 20 THEN
        Write '10 ≤ y < 20'
      ELSE
        Write '20 ≤ y'

```

```

IF y < 0 THEN
  Write 'y negative'
ELSE IF y < 5 THEN
  Write '0 ≤ y < 5'
ELSE IF y < 10 THEN
  Write '5 ≤ y < 10'
ELSE IF y < 20 THEN
  Write '10 ≤ y < 20'
ELSE
  Write '20 ≤ y'

```

Please keep in mind that the conditions are checked one after the other, so the solution sets need not be disjoint, i. e. they are allowed to overlap.

(2.37) In the **multiple selection with selector value**, the decision which branch will be executed depends on the value of an expression, the so-called selector value. Each branch ends with the beginning of the following branch.

Note The multiple selection with selector value is supported by the language Pascal (and Basic), but not by the language C++ or C; but in these latter languages there is a construct that can be used for imitating the multiple selection.

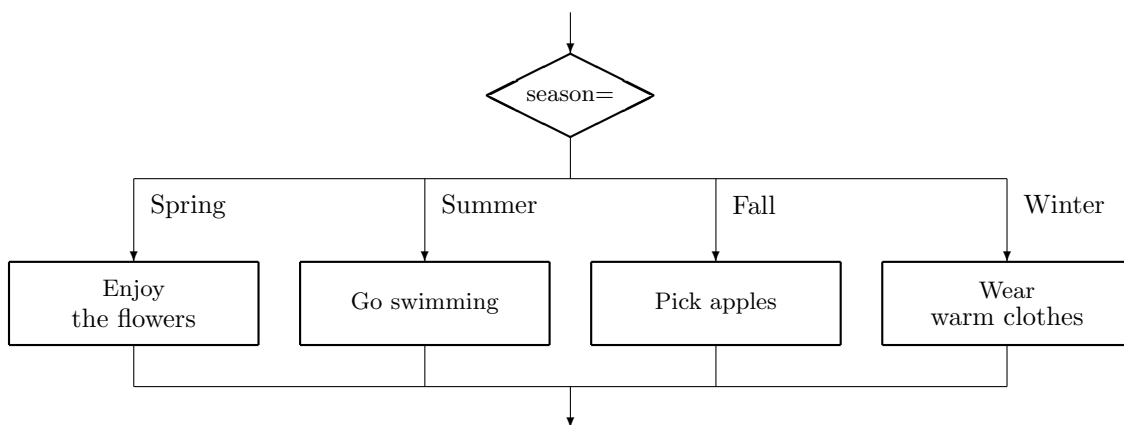
(a) Pseudo-code:

```

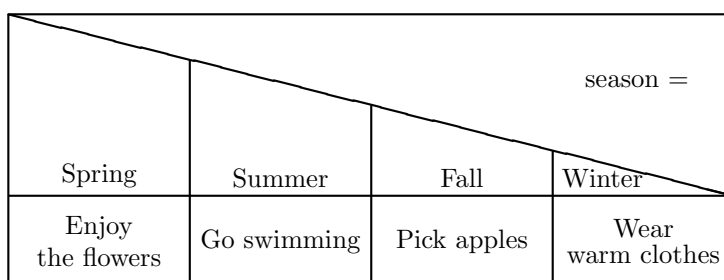
CASE DISTINCTION ACCORDING TO season:
  Spring:  Enjoy the flowers
  Summer: Go swimming
  Fall:    Pick apples
  Winter:  Wear warm clothes

```

(b) Program flow chart:



(c) Structure chart:

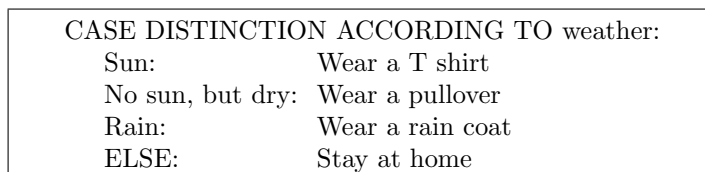


(2.38) Sometimes it is helpful to have an ELSE branch.

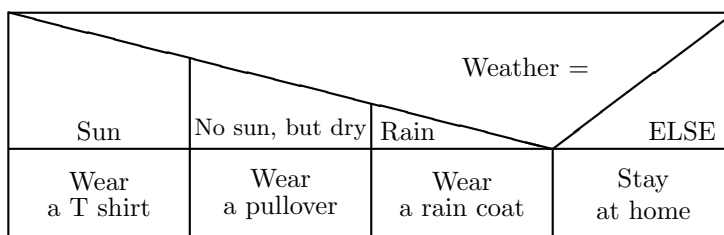
Note Not all languages that support the multiple selection implement the ELSE branch.

Ex for a multiple selection with an ELSE branch:

(a) Pseudo-code:



(b) Structure chart:



(2.39) You can see in the pseudo-code that a processor that wants to realize the selection structure must be able to perform forward jumps within the instruction text, i. e. to omit instructions in forward direction, see (2.26).

2.4 Iteration

(2.40) **Ov** In an **iteration** (Wiederholung, Iteration)—that is the second of the control structures (2.13a)—, a statement, the so-called **loop statement**, will be executed several times.

At first, you can distinguish two kinds of loops:

- Checking the entrance condition before entering the loop:
pre-tested loop or head-controlled loop; here it is possible that the loop will never be executed.
- Checking the condition after running through the loop:
post-tested loop or foot-controlled loop; this loop will be executed at least once.
Depending on the language, there are two representations of the post-tested loop:
 - expressing the condition as a re-entry condition,
 - expressing the condition as an exiting condition.

These two conditions are logically inverse to each other.

Note In this course, the post-tested loop will always be used with the re-entry condition because this one is supported by C++/C; the exiting condition corresponds to the language Pascal.

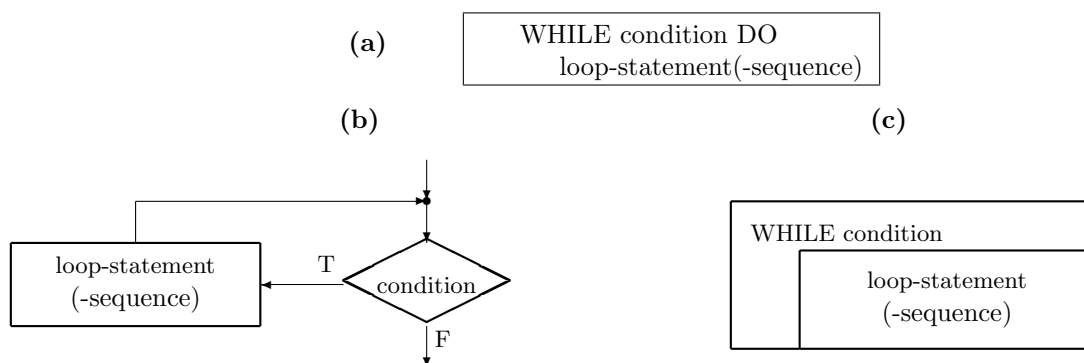
In all of these kinds of loops, the number of loop executions is not necessarily settled when first entering the loop.

There is another kind of loop; here the number of repetitions is settled when entering the loop: the **counting loop** (2.44).

- (2.41) The **pre-tested loop** (kopfgesteuerte Schleife, or: abweisende Schleife) will be performed in the following way: First the condition will be checked: if it is FALSE, the loop will never be entered, if it is TRUE, the loop will be executed once. After this, the condition will be checked once more; if the condition is TRUE again, the loop will be executed once more etc. Students (not only beginners) 'like' to create infinite loops by expressing the condition in a wrong way: if it will never be FALSE, the loop will be executed an infinite number of times. Therefore it is important that the loop condition must depend in some way on running through the loop. (Another way of changing the condition is that it depends on an outside event, e. g. pressing a key—not used in this course.)

The pre-tested loop in the three representations:

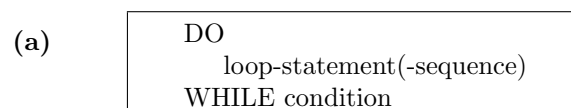
- (a) pseudo-code, (b) program flow chart, (c) structure chart:

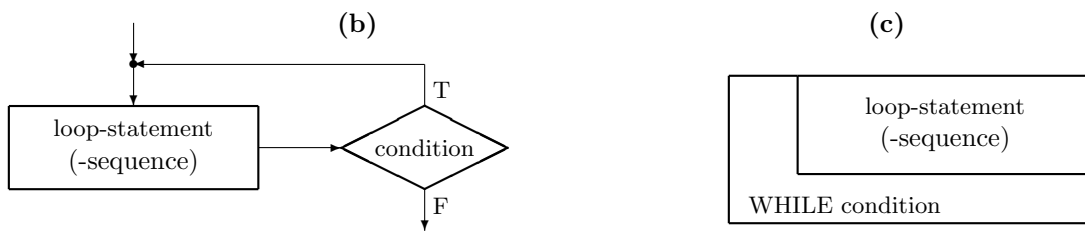


- (2.42) The **post-tested loop** (fußgesteuerte Schleife, or: nicht-abweisende Schleife), expressed in the manner of C++/C as a re-entry condition, will be interpreted in the following way: At first, the loop will be executed once. Then there is a check to see if the loop shall be executed once more, i. e. if the re-entry condition has the value TRUE. After the next loop execution this condition will be checked once more etc. When writing this loop, there is the risk of an (unwanted) infinite loop, too.

The post-tested loop (C++/C) in the three representations:

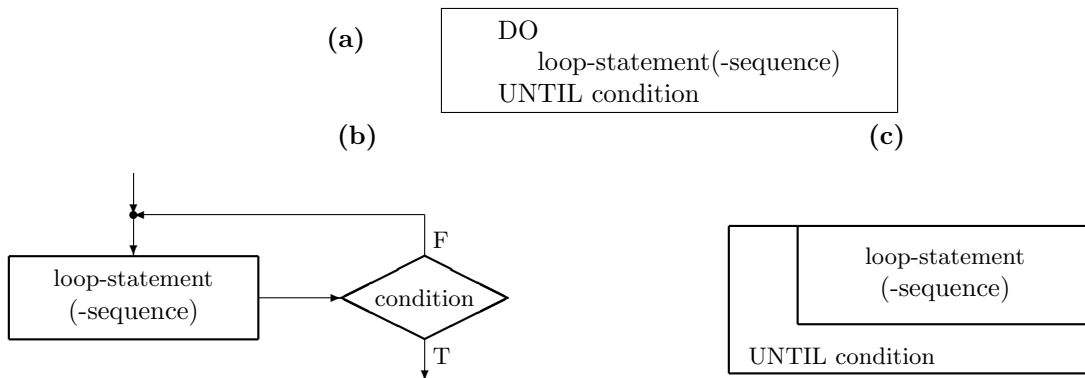
- (a) pseudo-code, (b) program flow chart, (c) structure chart:



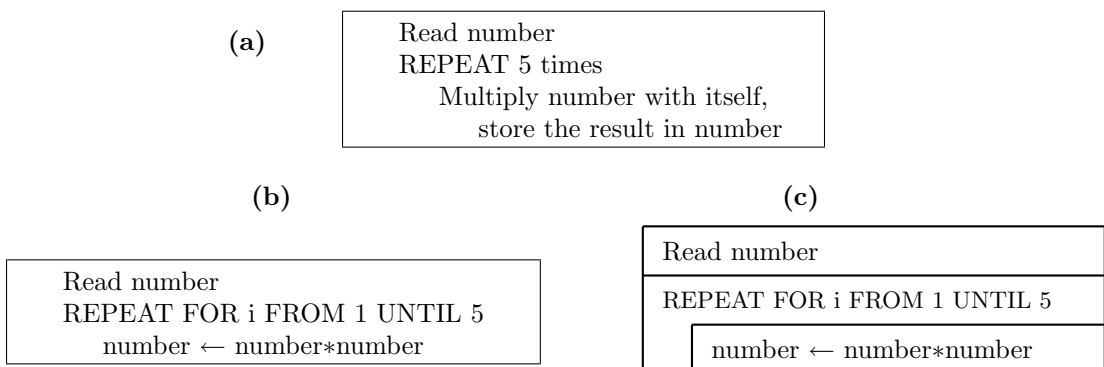


Note In this pseudo-code, the keyword *WHILE* is used, just as in the pre-tested loop. Here, however, there is no grammatical conflict as within the selection structure (2.34↑↑H), the grammar is unambiguous. However, if you will have only a quick look at the algorithm, you easily can make a misinterpretation, i. e. confusing pre- and post-tested loop. The risk of confusing these two loops will be accepted here because it also exists in the languages C++/C. Consequently, the recommended non-indentation of *WHILE*—in contrast to (2.22b)—can make some problems; how to solve the problem in C++/C see (ch. 4.7).

(2.43) The post-tested loop in the Pascal manner (condition expressed as an exiting condition) in the three representations:



(2.44) The **counting loop** is realized mostly with a loop variable counting the number of loop entrances. In (a) you will find a counting loop without such a variable, in (bc) with such a counting variable.



(2.45) You see in the pseudo-code that in order to realize an iteration structure, a processor must in addition be able to perform backward jumps as well as performing forward jumps, see (2.26).

2.5 Nesting the three algorithmic structures

(2.50) **Ow** These three algorithm structures sequence, selection, and iteration can be nested to an arbitrary depth. (Nesting a sequence within a selection or an iteration is already indicated in former diagrams.)

You may build arbitrarily complicated algorithms solely by using these three structures, together with the ability to nest them. As mentioned earlier (2.13b), these basic structures are sufficient; especially there is no need to use direct jump statements (‘goto’). Such jump statements are prohibited for students in this course; therefore they will not be introduced in C++.

Theoretically, the nesting may be performed into arbitrary depth. But there is a limit important for the practice, i. e. the demand that the human reader must be able to grasp the algorithmic structure, in any case. Therefore, in this course much stress is laid on the **layout** of the program text (as pseudo code or in a programming language). The basic rules are explained in the following point (2.51); the same rules will be applied in more details on C++ in (4.73).

The items (2.52) and (2.53) give examples for nesting. The second example, the algorithm ‘bubble sort’, may be needed for later applications; learning by heart is not necessary, remembering and finding it again is sufficient.

(2.51) Rec —for students: this paragraph describes a must!!—

Begin and end of the different nesting levels must be observed very strictly. In the pseudo-code this will be done by indentations, see (2.52a).

During programming, it is very important to have so much self discipline that each **nesting level indentation** will be made very accurately—for each nesting level a definite indentation distance. This is important only for the human user! A C++ compiler ignores indentations because the language allows an arbitrary layout.

Also here, (2.22b) continues to be valid: Each **statement** (normally) should be written in one line. If it is too long, the statement should be **continued** in the next line(s) with **indentation**.

Exceptions, i. e. no indentation despite continuing a statement (i. e. a control structure statement):

- Two branch selection: ELSE directly under IF, not indented (2.32, 2.34 ↑↑),
- IF-ELSEIF chain (2.36),
- Post-tested loop: WHILE directly under DO, not indented—but it has some problems (2.42 *Note*); in C++/C see (4.73).

In structure charts, the nesting levels will be represented by graphically nesting one rectangle into another, see (2.52c).

(2.52) Ex The nesting levels of the following example:

In the uppermost (first) nesting level, there are—beside the description of what memory is needed—three statements, in total. The second statement has a complex structure, it is a two branch selection. In the next nesting level, you see that both the THEN branch and the ELSE branch consist of a two branch selection (each in the second nesting level). The assignment statements belong to the third nesting level.

(a) Pseudo-code—left with normal indentations, right with extra indications of the nesting levels:

```

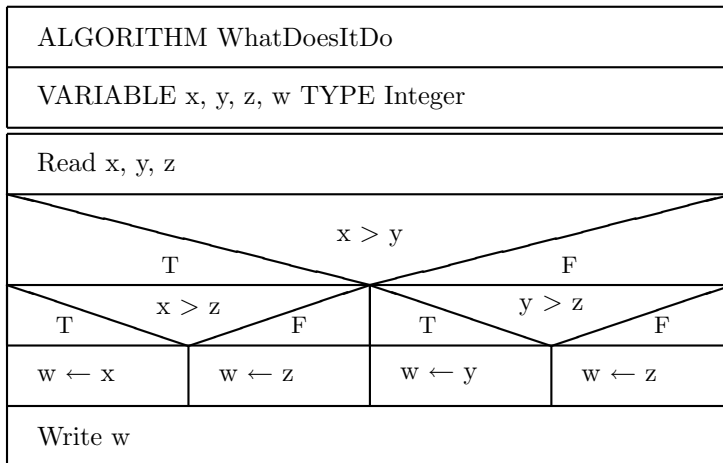
ALGORITHM WhatDoesItDo
BEGIN
  VARIABLE x, y, z, w TYPE Integer
  Read x, y, z
  IF x > y THEN
    IF x > z THEN
      w ← x
    ELSE
      w ← z
  ELSE
    IF y > z THEN
      w ← y
    ELSE
      w ← z
  Write w
END
    
```

```

ALGORITHM WhatDoesItDo
BEGIN
  VARIABLE x, y, z, w TYPE Integer
  Read x, y, z
  IF x > y THEN
    IF x > z THEN
      w ← x
    ELSE
      w ← z
  ELSE
    IF y > z THEN
      w ← y
    ELSE
      w ← z
  Write w
END
    
```

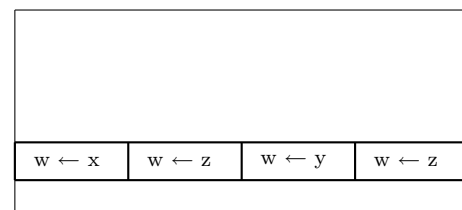
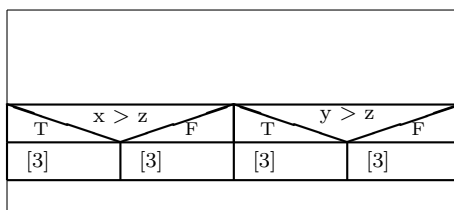
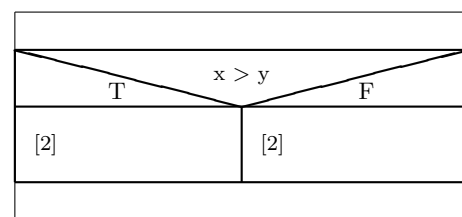
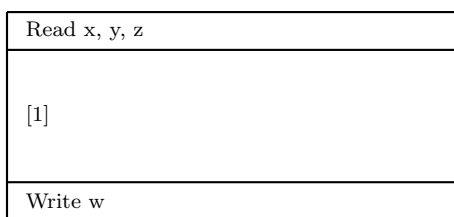
(b) Program flow chart: independent exercise, please; explanation later on in the lecture, if needed.

(c) Structure chart:



The individual nesting levels:

[n] shall mean: statement in the nesting level n; the statements marked in this way are each shown in the following picture with their inner structure.



(2.53) Algorithm for sorting:
Bubble-Sort

(Sortieren durch Nachbarvertauschungen (not literally translated)).

The type

Integer-ARRAY[count] shall mean an array of count integer elements; they shall be numbered from 0 until (count-1).

Development of this algorithm and more explanations see lecture.

Exercise:

Draw a structure chart corresponding to the algorithm in the right.

<pre> ALGORITHM BubbleSort BEGIN VARIABLE num TYPE Integer VARIABLE swapped TYPE Boolean CONSTANT count TYPE integer VALUE ... VARIABLE numberArray TYPE Integer-ARRAY[count] Read count numbers, store them into numberArray DO swapped ← false num ← 0 WHILE num < count-1 DO num ← num + 1 IF numberArray[num-1] < numberArray[num] THEN Swap both numbers swapped ← true WHILE swapped Write numberArray END </pre>

(2.54) **Top-down-design** (Prinzip der schrittweisen Verfeinerung—literal English translation of these German words ‘stepwise refinement’): a kind of decomposition which is oriented to the flow of problem solution. Each refinement level will be decomposed into finer units until there are only **elementary algorithms**, i. e. algorithms which can be directly executed by the processor. Within each refinement level, you can use the three algorithm structures sequence, selection, iteration.

↑↑ Some authors say that the terms ‘stepwise refinement’ and ‘top-down-design’ are not synonymous.

2.6 Operators, logical combinations**(2.60)** Ovw At first, this subchapter covers the explanation of **operators** (i. e. symbols for functions) and their applications (2.61). Then special expressions are considered in more details, to be precise the logical expressions or **propositions** (they may result only in ‘true’ or ‘false’). Especially the **operators belonging to them** are introduced (2.62ff.). In the following chapters, these terms will be applied to the language C++; they will be essential for comprehension.**(2.61)**

(a) An **operator** (Operator) is a symbol for a function, i. e. for an unambiguous mapping of one or several elements, the so-called **operand(s)** (Operand(en)), to the result value.

(b) An **expression** (Ausdruck) can consist of:

- a constant (without an operator or as an operand), or
- a variable (without an operator or as an operand), or
- an operator with its operands, or
- all these parts possibly nested in arbitrary depth.

(c) You can distinguish operators e. g. by the **number of operands**:

(1) An operator with one operand is called **unary** (unär, monadisch).

Different notations with respect to the order of operator and operand (the operator symbol shall be \otimes , the operand a):

$\otimes a$	prefix (präfix)	Ex	$f(a)$, $-a$ (sign as an operator), $\neg p$ (log. inversion, see (2.63))
$a \otimes$	postfix (postfix)	Ex	$4!$ (factorial)

(2) An operator with two operands is called **binary** (binär, dyadisch).

Notation (a, b shall be operands):

$\otimes ab$	prefix	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>Ex</td><td>$f(a, b), \quad +(3, 4)$</td></tr></table>	Ex	$f(a, b), \quad +(3, 4)$
Ex	$f(a, b), \quad +(3, 4)$			
$a \otimes b$	infix (infix)	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>Ex</td><td>$5 + 6, \quad 4 \times 3$</td></tr></table>	Ex	$5 + 6, \quad 4 \times 3$
Ex	$5 + 6, \quad 4 \times 3$			
$ab \otimes$	postfix	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>Ex</td><td>$5\ 6\ +$, cp. some HP pocket calculators ('inverse Polish notation')</td></tr></table>	Ex	$5\ 6\ +$, cp. some HP pocket calculators ('inverse Polish notation')
Ex	$5\ 6\ +$, cp. some HP pocket calculators ('inverse Polish notation')			

Sometimes a binary operator will not be written, e. g. ab for $a \times b$ (or $a \cdot b$), or only by different baselines of the operands, e. g. a^b for the operator power.

- (3) An operator with three operands is called **ternary** (ternär).
- (d) If there are different operators in an expression, the so-called **precedences** or **hierarchy levels** (Vorrangregeln, Hierarchiestufen, auch: Prioritätsstufen) must be defined. Operators with higher precedence bind their operands more strongly than those with lower precedence. These bindings can be overridden by explicit use of parentheses ().

Ex

 Operators of different precedence: $7 + 5 \times 3$ ('multiplication before addition')—in contrast to using parentheses: $(7 + 5) \times 3$.

- (e) If there are several operators of the same precedence or the same operator multiple times, the **associativity** (Assoziativität, Bindungsrichtung) must be settled, i. e. the direction of combination: **left associative**, i. e. combining from left to right, or **right associative**, i. e. combining from right to left (more seldom, in C++/C in some precedence levels).

(2.62)

- (a) Definition of the terms **proposition** and **truth value** see (2.31).
The truth values TRUE und FALSE shall be abbreviated with T and F.

Ex	The sun is shining here and now.	No student will learn computer science.	$1 = 2$	$16 + 3 \times 7$	Please, come to me!	Is it raining now?	$(a + b)^2 = a^2 + 2ab + b^2$	$16 > 17$	On January 1, 2010 at 12:00, the sun is shining in New York.	proposition?	truth value, if proposition?
----	----------------------------------	---	---------	-------------------	---------------------	--------------------	-------------------------------	-----------	--	--------------	------------------------------

Abbreviation for propositions (often): Latin letters (p, q, r, \dots).

- (b) The TYPE of variables that represent truth values is called 'logical' or 'Boolean' (after Boole, British mathematician, see term 'Boolean algebra', i. e. algebra with Boolean values).

(2.63) Applied to propositions, there are four different **unary operators**, in total.

↑↑

There are two different starting values T and F, each value can be mapped to two different values, so there are 2^2 different operators.

Interesting—besides the identity—is the operator **Not**, or negation, or (logical) inversion: it inverts the truth value of its operand.

Symbol: $\neg p$, or \bar{p} (or $\sim p$).

Truth table (Wahrheitstabelle):

p	$\neg p$
T	F
F	T

(2.64) Applied to propositions, there are 16 different **binary operators**, in total.

↑↑

There are four different starting value pairs (T,T), (T,F), (F,T), (F,F); each pair can be mapped to two different results, so there are 2^4 different operators.

- (a) Four of these operators are important here:
 - **conjunction, And** (Konjunktion, Und), symbol: \wedge or AND
 - **disjunction, inclusive Or** (Disjunktion, inklusives Oder), symbol: \vee or OR
 - **non-equivalence, exclusive Or** (Antivalenz, exklusives Oder), symbol: $\succ\prec$, or XOR, or \oplus
 - **equivalence** (Äquivalenz), symbol: \longleftrightarrow

The truth tables:

p	q	$p \wedge q$	$p \vee q$	$p \text{ XOR } q$	$p \longleftrightarrow q$
T	T	T	T	F	T
T	F	F	T	T	F
F	T	F	T	T	F
F	F	F	F	F	T

The **And** relation is T if and only if both operands are T.


The **Or** relation is F if and only if both operands are F.

The **non-equivalence** or **exclusive Or** relation is T if and only if both operands have different truth values. In English more exactly: either ... or ...

The **equivalence** relation is T if and only if both operands have the same truth values.

(b)

- (1) All four operators from (a) are—without the short-circuit evaluation (b2)—**commutative** and, if unmixed, **associative**.
- (2) In some programming languages, the so-called **short-circuit evaluation** will be applied to the operators **And** and (inclusive) **Or**: the right hand operand of an And or Or operator will be evaluated only if the result value has not yet been determined by the left hand operand alone.
 - The result value T is settled for an Or expression if the first operand is T; this result value does not depend on the value of the second operand.
 - The result value F is settled for an And expression if the first operand is F; this result value does not depend on the value of the second operand.

If the short-circuit evaluation is applied, both the operators And and Or are **not commutative** any more —in contrast to (b1).

Ex The precedences of the operators used in this example shall be (from higher to lower precedence, within each line the same precedence):

$$\begin{array}{l} / \quad \quad \quad \text{(division)} \\ > \neq = \quad \quad \text{(= shall be the operator for check on equality)} \\ \wedge \quad \vee \end{array}$$

The expression $a \neq 0 \wedge b/a > 3$ can always be evaluated only if the short-circuit evaluation is guaranteed. Otherwise, the processor could possibly have to perform a division by zero, i. e. if $a \neq 0$ equals F.

The expression $a = 0 \vee b/a > 3$, too, can always be evaluated only if the short-circuit evaluation is guaranteed. Otherwise there can be a division by zero, i. e. if $a = 0$ has the value T.



It is very difficult to understand the meaning of the program text if the right hand operand has additional effects ('side effects', (ch. 3.3)), e. g. in the following expression:

$b > 0 \wedge (\text{readNumber}() < 10)$

(readNumber() shall read a number and return its value),

because: if $b > 0$ is T, the number would be read, if it is F, it will not be read.

Note The languages C++/C guarantee the short-circuit evaluation with the operators And and Or; a Pascal compiler is allowed to do so too, but it need not.

(c) De Morgan laws:

$$\overline{p \wedge q} \longleftrightarrow \overline{p} \vee \overline{q}$$

$$\overline{p \vee q} \longleftrightarrow \overline{p} \wedge \overline{q}$$

The inversion of an And relation results in an Or relation of the inverted operands, and vice versa.

- (d) **Simplification** of complex Boolean expressions can be performed by applying logical rules (see books about formal logic) or by Karnaugh-Veit (KV) maps (see books about digital electronics).

3 First steps with C++

3.0 Overview

This chapter aims at a first approach to very simple C++ programs. Here you find the beginnings of the C++ syntax explanation, in addition the basic data types. Subchapter 3 renders possible effects of expressions at run time (main effect, side effect); surely, the frequent use of these terms is a speciality of this course (in contrast to nearly all textbooks).

3.1 The first programs

(3.10) Ow After introducing two very simple, but complete C++ programs, the formal aspect of these programs, i. e. the syntax, is described. It will simplify your further learning very much, if you familiarize yourself straight away with this kind of syntax description (a modified BNF notation, see (ch. 0.1)), and if you comprehend the formal aspect of the example programs with the aid of the syntax description. In the lecture, this will be explained in more details than is written here.

(3.11) // My first program in C++ (example file E03-11.CPP)

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello, world!" << endl;
    return 0;
}
```

(3.12) Another program, see (2.25d):

// Calculate the volume of a circular cone (example file E03-12.CPP)

```
#include <iostream>
using namespace std;

int main()
{
    double radius, height, volume;
    const double pi=3.14159;

    cin >> radius >> height;
    volume=radius*radius*pi*height/3;
    cout << "Volume: " << volume << endl;

    return 0;
}
```

(3.13)

(a) *C++ program* _[NC] \doteq
 $\{ \textit{IncludeLine} \}_{0..n}$
 using namespace std;
 int main()
⁵⁴*CompoundStatement*

Frequently used include line: #include <iostream>

Note `IncludeLine(s)` do not really belong to a C++ program, because they are executed by the preprocessor before compilation (5.74). The sense of the `using` directive is briefly indicated later on (8.23c).

(b) $^{54} \text{CompoundStatement} \doteq \{ \text{Statement}_{0..n} \}$

$^{50} \text{PartStatement} \doteq$
 $\text{CompoundStatement} \mid \text{ExpressionStatement} \mid \text{DeclarationStatement}$

$^{51} \text{ExpressionStatement}_{\text{[NC]}} \doteq$
 $\text{Variable} = \text{Expression} ; \mid$
 $\text{C++} \quad \text{cout} [\ll \text{Expression}]_{1..n} ; \mid$
 $\text{C++} \quad \text{cin} [\gg \text{Variable}]_{1..n} ;$

Note 1 Semantics of the assignment expression (above, first alternative): The expression on the right hand side of the assignment operator `=` will be evaluated, and its value will be assigned to the variable on the left hand side of the operator. Therefore, the direction of thinking here is from right to left—similar to solved equations in mathematics or natural sciences.

Note 2 The output on the screen is made with a `cout` statement (above, second alternative). To produce a new line, you may use the special expression `endl`. ↗ Details see (5.22).

Note 3 The input from keyboard is done by a `cin` statement (above, third alternative). ↗ Details see (5.21).

Examples for $^{57/11} \text{etc. DeclarationStatement}$:

```
int VarName ;
double OneMoreVarName ;
double VarName1, VarName2 ;
```

Note 4 During run time, the above `DeclarationStatements` result in reserving memory space for an `int` variable and for three `double` variables with the cited name. Details about the types `int` and `double` see (ch. 3.2).

(c) *Name, Identifier* $\langle \text{Name, Bezeichner} \rangle \doteq \text{Letter} [\text{Letter} \mid - \mid \text{DecimalDigit}]_{0..n}$

Note `C/C++` is case sensitive, i. e. it distinguishes between capital and lower case letters; please pay attention!

Rec The underscore is indeed allowed at the beginning of a name, but you should not use it because otherwise there could be conflicts with compiler internal names.

Letter \doteq A | B | C | ... | Z | a | b | ... | z
DecimalDigit \doteq 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

(3.14) Comments, are ignored by the compiler, but are important for the human reader's clarity:

```
C++ // ... end line comment
C/C++ /* ... */ general comment, possibly over several lines or only part of a line
```

(3.15) About the statement `return 0;` at the end of the program:

Return with value 0 (often with the meaning: 'without errors') to the process that invoked the C++ program, normally to the operating system. Its omission was ignored by some older compilers, yet this was not correct; in `[Forb]` `C++(new)` this statement will be automatically added if needed.

3.2 Simple data types, strings, operators

(3.20) **Ow** (Data) type: a name or a description for a set of values with allowed operations belonging to them, see remarks about `TYPE` in (2.25a2).

In this subchapter, simple data types (i. e. not compound ones) are described. It is important that you comprehend the basic principle of naming and meaning of as many types as possible, furthermore that you get (and retain!) an idea of the number ranges belonging to them. In this context, it is valuable to have a look at (ch. 1.4) for repetition. In addition, the most important operators that may be applied to these basic types are explained briefly. Without further type explanation (which will be presented much later), the string constant is introduced.

(3.21)

(a) Examples for simple data types:

- Integer numbers:
 - 'characters', mostly very small integer numbers:
 - `char`
 - `signed char`
 - `unsigned char`
 - small integer numbers (seldom used):
 - Synonyms: `short int`, `short`, `signed short`, `signed short int`
 - Synonyms: `unsigned short int`, `unsigned short`
 - 'normal' integer numbers:
 - Synonyms: `int`, `signed int`, `signed`
 - Synonyms: `unsigned int`, `unsigned`
 - large integer numbers:
 - Synonyms: `long int`, `long`, `signed long int`, `signed long`
 - Synonyms: `unsigned long int`, `unsigned long`
 - ⚠ Take care with unsigned types: great risk (type conversions, range overflow), see (1.44c)!! ↑↑ Details see (C++/ch. 7).
- Floating point numbers (in increasing accuracy):
 - `float`
 - `double` (as 'normal' floating point type)
 - `long double`

(b) Memory space, number range

type	memory space (normally)	signed (Assumption: two's complement)	unsigned
<code>char</code>	1 byte	-128...127	0...255
<code>short int</code> , <code>int</code> (Note1)	2 bytes	$-2^{15} \dots 2^{15} - 1$ (-32 768...32 767)	$0 \dots 2^{16} - 1$ (0...65 535)
<code>long int</code> , <code>int</code> (Note1)	4 bytes	$-2^{31} \dots 2^{31} - 1$ (-2 147 483 648...2 147 483 647)	$0 \dots 2^{32} - 1$ (0...4 294 967 295)

Note1 The type `int`—as the standard integer type—normally is a 2 byte number (as above `short int`) or a 4 byte number (as above `long int`), depending on the implementation.

Note2 The above mentioned memory spaces are minimum spaces; the implementation is allowed to take broader representations.

Note3 `signed char` is signed, `unsigned char` is unsigned; `char` however, depending on implementation or processor, is signed (mostly) or unsigned (more seldom). All three types are regarded as different types.

In compilers for Intel processors, the type `char` is normally signed.

Note4 The three `char` types have different behavior according to context; arithmetically, they are small integer numbers, for input and output (with `cin/cout`) they are characters.

(3.22) $IntegerConstant \doteq [\begin{smallmatrix} + \\ - \end{smallmatrix}]_{opt} DigitS\ Suffix_{opt}$

$DigitS \doteq DecimalDigitSequence$

$Suffix$: `u` or `U`: unsigned, `l` or `L`: long, both: unsigned long,
missing: `int`, if matching to the number range

↑↑ Details see (C++/7.5)

Interpretation (ONLY of an integer number, not of a floating point number):

- decimal if without a leading zero,
- ⚠ octal if with leading zero,
- hexadecimal if with leading `0x` or `0X`.

(3.23) $FloatingPointConstant \doteq$

$[\begin{smallmatrix} + \\ - \end{smallmatrix}]_{opt} [\begin{smallmatrix} DigitS \cdot DigitS_{opt} \\ DigitS \end{smallmatrix}] ExpPart_{opt} Suffix_{opt} \mid [\begin{smallmatrix} + \\ - \end{smallmatrix}]_{opt} DigitS ExpPart Suffix_{opt}$
 $ExpPart \doteq [\begin{smallmatrix} e \\ E \end{smallmatrix}] [\begin{smallmatrix} + \\ - \end{smallmatrix}]_{opt} DigitS$

Suffix: missing: double, f or F: float, l or L: long double

- (3.24) Operators—terms see (2.61c):
- unary prefix for integer and floating point numbers:
+ - (Op3ef: sign operators)
 - binary infix for integer numbers:
* / % (Op5abc: multiplication, division, remainder)
+ - (Op6ab: addition, subtraction)
 - binary infix for floating point numbers:
* / (Op5ab: multiplication, division)
+ - (Op6ab: addition, subtraction)

Hierarchy: a smaller Op number (e. g. Op5) binds more strongly than a larger number (e. g. Op6); these numbers refer to the hierarchy levels, general overview see (C++/ch. 2) and (5.11).

To override precedence levels, parentheses are used.

Ex $5 + 7 \times 3$ means: the product of 7 and 3 will be added to 5.
 $(5 + 7) \times 3$ means: the sum 5+7 will be multiplied by 3.

Implicit type conversions with binary operators:

If both operands do not have the same type, one of the types will be converted into the type of the other, the result is of the converted type; direction of conversion: integer to floating point number, smaller type to larger type.

↑↑ More precise rules see (C++/ch. 7) – ⚠ with unsigned types!!

- (3.25) *CharacterConstant* \doteq ' *Character* '
StringConstant \doteq " *Character*_{0..n} "
Character: (nearly) each representable character (including space), additionally:
 \ " instead of " (esp. in *StringConstant*)
 \ ' instead of ' (esp. in *CharacterConstant*)
 \\ instead of \
 \n as 'newline character'—independent from the internal representation
 within the operating system
 and some additional 'characters'

- (3.26) Special case of the *DeclarationStatement*:
ConstantDefinition _[NC] \doteq **const** *Type* *ConstantName* = *ConstExpression* ;

Note An initialization **MUST** take place! An assignment later on is not possible, it would contradict the constantness!

- (3.27) Compound assignment operators, see Op16b-k:
 Meaning: $a \text{ op} = b$
 is equivalent to: $a = a \text{ op } (b)$ with *op* one of the mentioned binary operators.

3.3 Expressions, side effects

- (3.30) Ov Presumably, this subchapter will seem to be difficult even after the (first) explanation in the lecture. You find the term **side effect** in nearly all C++/C textbooks, but probably not the term **main effect** (as the corresponding effect). However, when you have grasped that 'main effect' is just another name for 'value (of an expression)', and that 'side effect' means any other effect possibly appearing at run time, then you will recognize that many specialities of the languages C++/C may be described much more simply, which—without these terms—may be explained in a much more complicated manner.

↗ *Simpler description (if 'main effect' and 'side effect' are comprehended) e. g. at the following places (please, do not follow at the first read, but at later repetitions, in this case it is valuable!):* (2.64b2 Ex, 3.32b,d,e, 4.31c, 4.43, 5.12, 5.13, 5.21 Note3, 5.22 Note3, 5.24d, 5.25c) and other places.

- (3.31) In C++/C, an expression is defined by (definition not complete, cp. (2.61b)):

$$\begin{aligned}
 \textit{Expression} &\doteq \\
 &\textit{Constant} \mid \textit{Variable} \mid \\
 &(\textit{Expression}) \mid \\
 &\textit{FunctionCall} \text{ (7.12)} \mid \\
 &\textit{UnaryPrefixOperator Expression} \mid \textit{Expression UnaryPostfixOperator} \mid \\
 &\textit{Expression BinaryOperator Expression}
 \end{aligned}$$

If you have an expression, you have (often) to distinguish between two effects:

- **main effect**—another term for the value of the expression (almost always present).

Ex Expressions with main effect (value), but without any side effect (let *i* be a variable):

```

34
i
(i-17)/5

```

- **side effect** (Seiteneffekt, Nebeneffekt)—additional effect(s) of the expression (not always present), e. g. changing the value of a variable, output on the screen, input from keyboard, calling a function.

This term ‘side ...’ does not mean that it describes an unimportant, not wanted effect; no, if expressions have side effects, they are wanted!

Ex Expressions with side effect (which can be made to an expression statement (3.32a) by a postponed semicolon):

```

cout << 17      (output)
i = 24          (assignment, i. e. change of a value in memory; let i be a variable)
cin >> i       (input)

```

(3.32) Ex

(a) ⁵¹ $\textit{ExpressionStatement} \doteq \textit{Expression}_{opt} ;$

- Main effect unimportant (will be discarded), therefore normally the sense of an expression statement is a side effect (often: assignment expression, function call), other use: as an empty statement (*Expression* missing) if a statement is necessary syntactically, but there is nothing to do.

(b) $\textit{AssignmentExpression} \doteq \textit{Variable} = \textit{Expression} \quad (= \text{Op16a})$

- main effect: value (and type) of the variable after the assignment,
- side effect: the variable gets the value of expression.

(c) $\textit{AssignmentStatement} \doteq \textit{Variable} = \textit{Expression} ;$

(Special case of the ⁵¹ $\textit{ExpressionStatement}$)

- main effect (of the assignment as a whole) unimportant, will be discarded,
- side effect important: changing the value.

(d) Assignment chain: $\textit{Var1} = \textit{Var2} = \textit{Var3} = 25 ;$

Associativity from right to left (Op16), therefore interpretation:

$$\textit{Var1} = (\textit{Var2} = (\textit{Var3} = 25)) ;$$

The main effect of the inner assignment statement $\textit{Var3} = 25$ is important because it represents the expression value for the assignment to *Var2*; the main effect of the assignment expression $\textit{Var2} = (\dots)$ is the expression value for the assignment to *Var1*. Only the main effect of the assignment expression $\textit{Var1} = (\dots)$ is unimportant, it will be discarded.

(e) Increment and decrement operator (unary prefix Op3ab and postfix Op2fg): ++ --

- side effect: changing the value of the operand by +1 or -1—independently from prefix or postfix position!!
- main effect (here the prefix and postfix position must be distinguished very precisely!!):
 - postfix: old value of the operand,
 - prefix: new value of the operand.

(3.33) ⚠ *The time of execution of side effects is not fixed (it is implementation dependent)!! Certainly, all side effects are executed if the statement has finished.*

⚠ *Please, do not formulate expressions the value of which will depend on the time or order of execution of side effects!*

4 Algorithm structures in C++

4.0 Overview

The three important basic algorithm structures described independently of any programming language in (ch. 2) are now transferred to the language C++. Here, you get to know all forms of **sequence, selection, and iteration** supported by C++. Without knowing this, you can hardly write a C++ program.

After an example program, extremely important basic rules are explained (subchapter 7) for writing programs that make reading much more easy, i. e. rules for **generating names** and for **layout**. These rules will be valid throughout this course and possibly also for following courses.

4.1 Sequence, scope within blocks

(4.10) Ov The term **block** in the sense of a compound statement is introduced. Very often you will encounter the pair of braces creating this structure. In this context, further terms (scope, hiding, and local names) are described; read (7.15) for more details.

(4.11) The sequence (capable of nesting) (ch. 2.2) will be realized in C++ by the block:
⁵⁴ *CompoundStatement* (Verbundanweisung), also *Block* (Block) C++ \doteq {*Statement*_{0..n}}

(4.12) Names declared within a block are called **local names**. Their **scope** (Gültigkeitsbereich) (region in which the name exists) extends from the declaration point to the end of this block.

A name is **hidden** in subordinate blocks by declaring it there once more; after exiting from such a subordinate block, the name in its old meaning (variable: with its old value) is available again.

↗ Detail see (7.15).

(4.13) **Local variables** (variables declared within a block, e. g. within `main()`) mostly have no particular initial value, but a ‘random’ bit pattern. Therefore these variables must get a defined value before a read access.

↗ More exactly: actually, the mentioned variables are automatic variables (8.13). Exceptions are objects with soundly built constructors (9.21), and static variables (8.12).

4.2 Boolean expressions, data type bool

(4.20) Ov In contrast to C and C++(old), in C++(new) there is the Boolean data type; it has only two possible values, i. e. TRUE and FALSE. You should use this type often when only two different values are needed—and not the substitute type `int` for C programmers.

The Boolean expressions and their combinations appear very often; they are necessary to formulate selection and repetition statements.

(4.21) Data types for condition or proposition, cp. (2.31, 2.62).

(a) C++(new) Additional data type `bool`; the set of values consists only of the constants `true` and `false`.

If necessary: implicit conversion of an arithmetical expression into `bool`, i. e. value unequal 0 into `true` and value equal 0 into `false`, cp. (4.23).

Conversely, an expression of the type `bool` will be converted if necessary into an integer of the type `int`, i. e. `true` into 1 and `false` into 0.

(b) On the other hand C C++(old): a Boolean expression has the type `int`, i. e. if TRUE the value 1, if FALSE the value 0.

(4.22) c/c++: Combination of numerical expressions into Boolean expressions:

Op8:	$NumExpression < NumExpression$	$NumExpression <= NumExpression$
	$NumExpression > NumExpression$	$NumExpression >= NumExpression$
Op9:	$NumExpression == NumExpression$	$NumExpression != NumExpression$
	(== test for equality)	(!= test for inequality)

⚠ A ‘favorite’ mistake is mixing up both the operators = and ==:

Op16a: = operator for assignment (and frequently for initialization, too)
Op9a: == operator for check on equality

(4.23) c/c++: Each numerical expression can be interpreted logically (Boolean), i. e. as *condition*; a value unequal 0 is regarded as TRUE, a value equal 0 as FALSE.

(4.24) c/c++: Boolean values (and logically evaluated numerical expressions) can be combined into further Boolean expressions (2.63, 2.64):

Op13:	$BoolExpr1 \&\& BoolExpr2$	logical And
Op14:	$BoolExpr1 \ \ BoolExpr2$	logical inclusive Or
Op3d:	$! BoolExpr$	logical inversion

Details about both the binary operators $\&\&$ and $\|\|$ (Op13/14):

Note1 Here evaluation according to the short-circuit evaluation is guaranteed, i. e. the right hand expression $BoolExpr2$ will be evaluated only if the value of the total expression is not yet settled by the left hand expression $BoolExpr1$. Because of the short-circuit procedure, both these operators are not commutative; in contrast, without a short-circuit evaluation, they would be commutative. Cp. also the general discussion in (2.64b2).

Note2 Exception of (3.33): It is guaranteed that all the side effects of $BoolExpr1$ are performed before—if necessary— $BoolExpr2$ is evaluated.

(4.25) An expression e. g. in the form

$12 < x <= 34$

(as usual in mathematics) must be altered into two expressions linked by the And operator:

$12 < x \&\& x <= 34$

Note It is a pity, but the expression $12 < x <= 34$ is in C++/C syntactically permissible, too, but the interpretation is completely different: since the relational operators are left associative, at first the Boolean value for $12 < x$ (i. e. **true/false**) is converted into **int** (i. e. 1/0) according to (4.21a); after this, $1 <= 34$ or $0 <= 34$ is calculated as result value (i. e. in any case always true).

4.3 Selection

(4.30) Ovw The control structure **selection** (ch. 2.3) is supported by the ⁵⁵*SelectionStatement*. In C++/C, there are the one branch and the two branch selections and the multiple selection as an IF-ELSEIF chain. The multiple selection with a selector value is not supported directly; but it may be imitated very easily with the *switch-statement*. Please, pay attention to the trap of a left out **break**!

(4.31)

(a) The **one and two branch selections** (2.34, 2.32) are represented by the ⁶⁰*if-statement*:

OneBranchSelectionStatement \doteq **if** (*Condition*) *Statement1*

TwoBranchSelectionStatement \doteq **if** (*Condition*) *Statement1* **else** *Statement2*

Pseudo code of these selection statements:

IF *Condition* THEN *Statement1*

IF *Condition* THEN *Statement1* ELSE *Statement2*

Note1 A ⁹⁹*Condition* can be represented by each Boolean (logical) expression; additional possibility see (4.51).

Note2 As ⁵⁰*Statement* you can again use each statement; consequently, statements can be nested to an arbitrary depth—cp. chapter 2.5! Details about solving ambiguities see also (b).

- Note3** Please take care: *Statement is found in singular, i. e. there has to be exactly one statement.*
- If you need several statements, they are enclosed into a block (4.11); this block has the syntactical effect of one statement.
 - If you do not need any statement, you can take an empty statement, e. g.:

```

; // expression statement without expression, (3.32a)
{ } // block with zero statements (4.11)

```

- (b) In pseudo code, we agree that if you want to nest a one branch and a two branch selection, the correspondence of the ELSE branch is indicated by indentation: in version A the ELSE branch belongs to the outer IF, in version B to the inner IF:

<pre> // Version A IF outer-condition THEN IF inner-condition THEN inner-then-statement ELSE outer-else-statement </pre>	<pre> // Version B IF outer-condition THEN IF inner-condition THEN inner-then-statement ELSE inner-else-statement </pre>
--	--

In contrast to this: since the C++ compiler works independently from the layout, the ambiguity is solved by the following rule:

An **else** always belongs to the last **if** which did not yet have an **else**.

Therefore version A and version B will be interpreted—by the C++ compiler—in just the same way; the ELSE branch belongs to the inner IF. In order to force the interpretation according to the layout of version A on the C++ compiler, the inner one branch selection must be masked so that it does not appear as a selection:

```

// Version C—necessary for the C++ compiler
IF outer-condition THEN
  STATEMENTBRACKET
    IF inner-condition THEN
      inner-then-statement
    END STATEMENTBRACKET
ELSE
  outer-else-statement

```

This clamping of statements is done by the *CompoundStatement* (4.11) in C++, i. e. by enclosing in a pair of braces { }.

- (c) **⚠** As *Condition*, an assignment expression is also permitted because it has a main effect (3.32b); beginners are strongly advised against this. Mostly, an assignment is written unintentionally instead of a comparison (‘=’ instead of ‘==’).

Ex

```

int i; cin >> i;
if (i = -1) cout << "Yes";
else cout << "No";

```

Independently of the input, always Yes will be outputted!

- (4.32) The IF-ELSEIF form of **multiple selection** (2.36) can be realized by the **if-else-if** chain (continuing test for conditions if necessary); **Ex** (4.73).

- (4.33) The **multiple selection** with a selector value (2.37) is not directly supported by C++ und C. But you can imitate this structure with the ⁶¹*switch-statement*.

```

switch-statement ≐
switch ( Expression ) {
  [ case ConstExpression : ]1..n Statement0..n break ;
  // repetition of the latter line arbitrarily often,
  // i. e. inclusion of the whole line in [ ... ]1..n
  [ default : Statement1..n ]opt
}

```

Semantics:

- Firstly, evaluation of *Expression*,
- then jump to the **case**-label with the same value if present,

- else jump to the label `default` if present,
- else at once continue after this *switch-statement*.

⚠ *The switch-statement in C++ and in C is ONLY a jump switch for jumping in, not for jumping out. To imitate a multiple selection you must not forget the jump out statement ⁵³`break`; !! Details about this statement see (4.44a).*

Ex Filtering out the numbers 0 until 3 and other inputs:

```
int num;
cin >> num;

switch (num) {
    case 1: cout << "one";
           break;

    case 2: // no break      <-- comment sensible as note for "NO break"
    case 3: cout << "two or three";
           break;

    case 0: cout << "zero";
           break;

    default: cout << "negative or greater 3";
}

```

Note In the ⁶¹*switch-statement*, it is permissible to use ⁹⁹*Condition* instead of an *Expression*, too, details see (4.51b).

4.4 Iteration

(4.40) Ovw In C++ and in C there are three different kinds of loops:

- two are pre-tested iteration statements (*while-statement*, *for-statement*),
- one is a post-tested iteration statement (*do-while-statement*).

The counting loop is not directly supported, but it can be imitated by the (much more powerful) *for-statement*.

In practice, the post-tested loop is applied seldom particularly since its layout may generate problems; the *while-statement* and the *for-statement* are used very often. In all three cases, the repetition part must consist of exactly one statement; if you need several statements, you have to transform them syntactically into exactly one statement, i. e. the block, by enclosing them with a pair of braces.

(4.41) ⁷⁰*while-statement* \doteq `while (Condition) Statement`
Pseudo code (2.41): `WHILE Condition DO Statement`

(4.42) ⁷¹*do-while-statement* \doteq `do Statement while (Expression) ;`
Pseudo code (2.42): `DO Statement WHILE Expression`

Note1 Also here, the expression is interpreted as a condition, i. e. Boolean; syntactical difference to *Condition* see (4.51b)—this is not possible here.

Note2 The *do-while-statement* can cause problems with respect to the layout, see (4.73), see also (2.42 Note). Therefore: ‘I tend to avoid do-statements.’ (Str3/ch. 6.3.3)

(4.43) The most powerful loop:

⁷²*for-statement* \doteq `for (Expression1opt ; Condition2opt ; Expression3opt) Statement`

Semantics:

- First, there is the evaluation of *Expression1* (**initialization**)
 - main effect unimportant (is discarded),
 - therefore only sensible if there is a side effect, e. g. assignment,
 - is executed exactly once at the beginning,
 - can be left out.
- Afterwards, there is the test on *Condition2* (**entrance condition**)

- main effect important:
if **true**, entrance into the loop statement *Statement* (pre-tested loop!),
else exiting the *for-statement*,
- missing: meaning **true** (infinite loop).
- After each loop execution, there is the evaluation of *Expression3* (**reinitialization**)
 - main effect unimportant (is discarded),
 - therefore only sensible if there is a side effect,
 - will always be evaluated after each loop, before (once more) *Condition2* is tested,
 - can be left out.

Note1 Therefore, a *for-statement* is equivalent to the following construct (the only exception is if the statement '**continue**;' is used, see (4.44b)):

```

Expression1 ;
while ( Condition2 ) {
    Statement
    Expression3 ;
}

```

Note2 Extension for *Expression1* see (4.51c): *SimpleDeclaration*.

Ex Program fragment for output of the numbers 0 to 99, twice:

```

int i;
for (i=0; i<100; ++i)
    cout << i << endl;
for (i=0; i<=99; ++i)
    cout << i << endl;

```

(4.44) Two important ⁵³*JumpStatements*:

- (a) **break** ;
- allowed in a ⁶¹*switch-statement* and in all ⁵⁶*IterationStatements*,
 - immediate jump out of the *switch/iteration statement*
—in a *for-statement* no reinitialization,
 - in an *IterationStatement* no test on condition.
- (b) **continue** ;
- allowed in all ⁵⁶*IterationStatements*,
 - immediate termination of the current loop execution,
 - (if *for-statement*;) reinitialization,
 - then test on the (reentrance) condition.

4.5 The *condition* in control structures, scopes with more recent compilers

(4.50) **Ovv** With newer compilers **C++(new)**, you have the possibility to define a variable within the condition part of control structures which is valid only within this control structure (4.51a,b). Sometimes this may be cleverly used to avoid multiple calculations of the same value; however, in practice this is very seldom.

More often in practice, you will encounter the declaration in the initialization part of a *for-statement* (4.51c). Unfortunately, **C++(old)** and **C++(new)** differ regarding the scope. (It is a curiosity that the compiler Microsoft Visual C++ 6.0 behaves as **C++(old)** although otherwise it obeys the rules of **C++(new)**.) The behaviour of **C++(new)** is very sensible because you are able to generate loop variables, which are independent of all other program code and thus do not generate any conflicts.

(4.51)

- (a) **c/c++** In the ⁶⁰*if-statement* and in all three ⁵⁶*IterationStatements* a *BooleanExpression* or a logically evaluated *NumericalExpression* can be taken as *Condition* or *Expression*.
- (b) **C++(new)** In the statements
- ⁶⁰*if-statement*,
 - ⁶¹*switch-statement*,
 - ⁷⁰*while-statement*,

- ⁷²*for-statement*,

i. e. in all control structures except the ⁷¹*do-while-statement* there is another possibility, the ⁹⁹*Condition*:

*TypSpecifier*_{1..n} *Declarator* = *Expression*

—in other words a definition of a variable with initialization.

The scope of this variable is the area only within the control structure.

This variable definition is not allowed in the ⁷¹*do-while-statement*; such a possibility would be senseless because the definition would have to occur after the use.

- (c) `C++` Additionally, a ¹¹*SimpleDeclaration* is allowed as the initialization part of a *for-statement*.

⚠ The scope of these variables is different depending on `C++(old)` or `C++(new)`:

- `C++(old)` normal introduction of a variable, scope also after the control structure,
- `C++(new)` scope only within the control structure, cp. (Str3/ch. 6.3.3.1).

⚠ *In this context, the compiler Microsoft Visual C++ 6.0 is regarded as `C++(old)`, probably as an exception.*

In `C++(old)` an imitation of the new behavior is possible simply by enclosing the whole control structure in a block.

4.6 Example

- (4.60) `Ovw` It is worth looking at the example program and to compare the algorithm line by line with the pseudo code (see no. 1 on the exercise sheet belonging to ch. 2, also available on the internet). If you want to run the program: like all example programs in the lecture notes, it is available on the internet, too.

- (4.61)

```
// Gregorian Calendar (example file E04-61.CPP)
// see exercises ch. 2, exercise 1
// Input as integer numbers: day month year
```

```
#include <iostream>
using namespace std;

int main()
{
    int day, month, year,
        yearH, yearR, help, weekDayNum;

    cin >> day >> month >> year;

    if (month < 3) {
        month += 10;
        --year;
    }
    else month -= 2;

    yearH = year/100;
    yearR = year%100;
    help = day + yearR - 2*yearH;
    help += (13*month-1)/5;
    help += yearR/4 + yearH/4;

    while (help < 0) help += 7;
    weekDayNum = help%7;

    switch (weekDayNum) {
        case 0: cout << "Sunday";    break;
        case 1: cout << "Monday";    break;
```

```

        case 2: cout << "Tuesday";   break;
        case 3: cout << "Wednesday"; break;
        case 4: cout << "Thursday";  break;
        case 5: cout << "Friday";    break;
        case 6: cout << "Saturday";  break;
    }
    cout << endl;

    return 0;
}

```

4.7 Recommendations for naming and layout, symbolic constants

- (4.70) Ovw To minimize errors in larger programs, it is essential that the names chosen by you explain themselves (‘self documenting’), furthermore—for the same reason—that you (nearly) always use constants as symbolic constants.

Additionally, the layout of the program text must be designed in such a way that a skilled programmer can unambiguously recognize the algorithm structure with only a quick glance at the text—in practice, there is no more time! It is essential that you practise these virtues immediately when you begin to learn programming so that obeying these rules will become natural for you. Therefore this is stressed very much in this course.

- (4.71) Rec **Naming convention and name spelling (lower and upper case)**

This is a must for the students!

- (a) Names which you select yourself should be ‘**self documenting**’, i. e. the names should express their purpose. Then you seldom need an explaining comment (in very large programs: only few explanations).

Ex Instead of `x` better `xValue`, instead of `t` better `time`, instead of `ts` better `startingTime` or `startTime`.

- (b) Recommendation for the **spelling** of self selected names (according to Booch/Rumbaugh/Jacobson, The Unified Modeling Language User Guide, 1998):

Each word part begins with a capital letter; the very first letter (i. e. the beginning of the first word part):

- of variables and functions (also inside of classes): a lower case letter,
- of types (e. g. class names) a capital letter.

Ex Variables and functions:

```

    pi, height, birthDay, nowMonth, weekDayNum, dayOfTheYear,
    newFunction, firstEntrance;

```

Types, class names:

```

    Date, CustomerArray, StudentNormal.

```

↑↑ Formerly—esp. in former C programs—the names of variables and functions were written mostly only with lower case letters, for separating the word parts the underscore was used, e. g.

```

    pi, height, birth_day, now_month, week_day_num, day_of_the_year,
    new_function, first_entrance;

```

New types and macros (ch. 5.6.5.7) were written mostly only with capital letters:

```

    DATE, CUSTOMER_ARRAY, STUDENT_NORMAL.

```

- (4.72) **Symbolic constants** are very important for good quality programming because they prevent ‘mystery numbers’. Mystery numbers hide their meaning, but symbolic constants emphasize it.

Ex Not everybody will guess that `6.28` may mean 2π ; with `twoPi`, however, this is clear. A later replacement of a more accurate value for `6.28` is nearly impossible; with a symbolic constant, this is very easy.

At best, symbolic constants are introduced by a constant definition (`const ...` (3.26)). Macros (5.72a) should be avoided; on the other hand, sometimes enumeration constants (12.21) may be used, see also summary in (10.11).


```
    if (...)
        Statement
    else if (...)
        Statement
    else if (...)
        Statement
    else
        Statement

// switch-statement
switch (...) {
    case ...: Statement break;
    case ...: // no break
    case ...: Statement
                Statement
                break;
    default: Statement
}

// while-loop
while (...) Statement
while (...)
    Statement
while (...) {
    Statement
    Statement
}

// for-loop
for (...) Statement
for (...)
    Statement
for (...) {
    Statement
    Statement
}

// do-while-loop (can cause problems)
// important: "while" not at the beginning of a line!
do Statement while (...);
do {
    Statement // should be a block even ...
    Statement // ... if only one statement
} while (...);

return 0;
}
```

5 Input and output, array and string type, supplements

5.0 Overview

This chapter renders a collection of very different topics that have to be described before introducing several programming styles in the following chapters (from ch. 6).

At first, the operators as a whole are presented (subch. 1); the table should often be consulted—also later on!

Subch. 2 explains the extremely important properties of the so-called streams. With the aid of these streams, in C++ the input and output are carried out. Since there is hardly a program without input or output, dealing with this is necessary.

Note *The application of input or output functions from C (5.21^{Note4}, 5.22^{Note4}) is not tolerated in this course because it is very error prone, and furthermore because the C++ streams can easily be handled as soon as there is comprehension of their behaviour.*

The generalization of the stream concept to dealing with text files is presented in subch. 3.

In subch. 4, the array type is introduced; it is used very often in applications for natural science and technology. It is applied to strings in the following subchapter.

A short introduction to types and especially their transformations (‘type casts’) is rendered in subch. 6; non-observance of such rules may generate surprising and hardly remediable errors.

In subch. 7, the preprocessor concept is explained which is used very often in C, but not so often in C++. However, there are some very important applications in C++, too.

5.1 Overview of operators

(5.10) Ov Here is a collection of all operators of C++/C. You will surely often have a look at the table even later on. For use in written examinations, there is a shortened version of this table in the notes ‘Language C++’ (ch. 0.2). It is important that—on the basis of this table—you learn and remember the basic operator structure (the extensiveness of which is hardly available in other computer languages).

After this, two new operators are presented: the conditional and the comma operator. These two operators are used very seldom or more often, depending on the personal programming style. You may abstain from them very often, but the operators may make a program more clear. Knowing how to use the comma operator in a *for*-loop head (5.13^{Ex}) is compulsory.

(5.11) List of all C++ **operators** with links if they are explained in this lecture (operators not explained here: ↑↑)

Operator levels 1 to 17: decreasing precedence (2.61^d)

– overriding the hierarchy level is possible by parentheses () –

Associativity (2.61^e): left associative →;

exceptions (‘@’, levels 3, 15, 16): right associative ←

Level	Operator	Meaning	Link	
1	a b	:: ::	<code>C++</code> unary prefix: global <code>C++</code> binary: scope resolution	(7.52), (9.12a)
2	a b c d e fg h i j k l	[] () () . -> ++ -- <code>static_cast<>()</code> <code>dynamic_cast<>()</code> <code>const_cast<>()</code> <code>reinterpret_cast<>()</code> <code>typeid</code>	index function call <code>C++</code> conv. SimpleTypeName member access member access via pointer postfix increment/decrement <code>C++(new)</code> type conversion <code>C++(new)</code> type conversion <code>C++(new)</code> type conversion <code>C++(new)</code> type conversion <code>C++(new)</code> type identification	(5.41) (7.12) (5.64b) (5.23a, 9.13) (10.29) (3.32e) (5.64a) ↑↑ ↑↑ (10.24 <small>Note4</small>), (12.63, 12.64) ↑↑
3@	ab unary prefix ef g h i j kl	++ -- ~ ! + - & * () <code>sizeof</code> <code>new delete</code>	prefix increment/decrement bit inversion logical negation sign address dereference type conversion memory size <code>C++</code> free store	(3.32e) (12.11) (4.24) (3.24) (10.21b) (10.21b) (5.64c) (5.14, 12.11) (10.31ff.)
4	ab	->* .*	<code>C++</code> member access	↑↑
5	a b c	* / %	multiplication division remainder	(3.24) (3.24) (3.24)
6	ab	+ -	addition, subtraction	(3.24)
7	ab	<< >>	bit shift streams: output, input	(12.11) (5.22, 5.21)
8	abcd	< <= > >=	comparison	(4.22)
9	ab	== !=	(un-)equality	(4.22)
10		&	bitwise And	(12.11)
11		^	bitwise exclusive Or	(12.11)
12			bitwise inclusive Or	(12.11, 5.33b)
13		&&	logical And	(4.24)
14			logical inclusive Or	(4.24)
15@	ternary	? :	conditional operator	(5.12)
16@	a bcde fgh ijk	= *= /= %= += -= >>= <<= &= ^= =	(simple) assignment compound assignment	(3.13b) (3.27)
17		,	comma operator	(5.13)

↑↑1 The hierarchy levels 15 (conditional operator) and 16 (assignment operators) are not defined unambiguously to each other, the precedence can be different, depending on context. Here, the order from C (K&R2/ch. 2.12) is rendered, it is compatible to the assignment of a conditional expression which is used very often (5.12Ex). In (Str3/ch. 6.2), the order is inverse.

↑↑2 Sometimes, also `throw` is presented as an operator (for the so-called exception handling), cp. (Str3/ch. 6.2).

(5.12) The **conditional operator** Op15 is a transfer of a two branch selection (2.32), (4.31) onto expressions:

$Expression1 ? Expression2 : Expression3$

(is equivalent to:) IF $Expression1$ THEN $Expression2$ ELSE $Expression3$

Semantics:

- First, $Expression1$ is evaluated as Boolean (ch. 4.2), additionally, also all the possibly existing side effects (3.31) of $Expression1$ are executed,
- afterwards:
 - either evaluation of $Expression2$ (if $Expression1$ is TRUE)
 - or evaluation of $Expression3$ (if $Expression1$ is FALSE),
- main effect of the total expression is either $Expression2$ or $Expression3$ —depending of which expression was evaluated.

Rec Since $Expression1$ is evaluated as Boolean, you may enclose it in parentheses () as a reminder.

Ex Assigning the absolute value of i to k : $k = (i >= 0) ? i : -i;$

(5.13) **Comma operator** Op17: $Expression1 , Expression2$

Semantics:

- First, $Expression1$ including all side effects is evaluated,
- afterwards evaluation of $Expression2$,
- main effect of the total expression: $Expression2$.

Application: if you need several expressions (with side effects) where only one is syntactically allowed.

Ex Often, this operator is used in the *for-statement* (4.43), e. g. if you need several loop variables:

```
int i,j;
for (i=0,j=10; j>0; ++i,--j) ...
```

 Other application: (5.26d modification).

(5.14) **Operator sizeof** (Op3j)

This operator returns the memory size (in bytes) of its operand. It can be used in two ways:

`sizeof(Type)`

`sizeof(Expression)`

It will be calculated: the memory size of one element of the type $Type$, or the memory size which $Expression$ would need if stored—not the value of $Expression$.

Ex – see also (5.55b) and (12.63a)

```
cout << sizeof(int)           // 4 with today's compilers
int i; cout << sizeof(i);     // also 4
cout << sizeof(13);          // also 4 (the constant 13 is an int)
```

⚠ The returned type is an **unsigned type**, therefore, pay attention if mixing with (normal) **signed types**, cp. (12.23 ⚠)! If the value is needed in an expression: first assign this value to an `int` variable, and use this variable afterwards.

⚠ If $Expression$ is an array name, see (5.55d) and (7.46 ↑↑). In any case safer: if you have an array, use of `ArrayType` is preferable to array name!

↑↑ When using $Type$, the parentheses () are syntactically necessary; when using $Expression$, they can be left. Therefore in the above example would be allowed the following, too:

```
sizeof i    ore also    sizeof 13.
```

But however, because the `sizeof` operator has a very high precedence level, using parentheses is strongly recommended also with $Expression$.

5.2 Standard streams, error handling

– see also (C++/ch. 3) –

- (5.20) Ov This subchapter introduces more details of the streams' world. Therefore, this will be explained in the lecture very intensively. A detailed knowledge of this field is compulsory for programming. You learn more details of the properties of the input and output objects (5.21, 5.22), after this the access via member functions and manipulators (5.23), with the aid of which comfortable formatting of the output can be performed.

It is very important to distinguish the two different kinds of reading: the read with the help of the input operator (>>) which at first discards leading white spaces (5.21 B_{ox}), and the input of each single character with the `get` function and related functions (5.24).

Point (5.25) renders an introduction to error handling by stream objects. Without knowing this, you are hardly able to write programs that run well, because incorrect input by the user always is possible. Since a stream object (usually) does not do anything in the case of an error, you must know that this case has to be handled and how to do this.

In (5.26) you find example programs (each as complete programs on the internet); understanding the effects of these programs is essential.

- (5.21) Variable name for the **standard input device/unit**: `cin` (character in)
 Standard input: mostly keyboard, can be redirected on the operating system level, see (1.35).
 'End of file character' if input from keyboard in DOS/Windows: Ctrl-Z (ASCII dec. 26), in Unix (depending on system setting) Ctrl-Z, too, or different, in Linux Ctrl-D as default.

Statement for reading: `cin [>> Variable]1..n ;`

Note1 More exactly, really [...]_{0..n}, but zero times normally senseless.

Note2 Main effect of the above expressions: see (5.25c) (stream object as a reference).

Note3 The input via `cin` is normally buffered by the OS (operating system):
 request C++ to OS or its keyboard buffer: in characters,
 reading from user by the OS into the OS keyboard buffer: in lines.

Using this buffered input via `cin`, the expression

`cin.rdbuf()->in_avail()`

returns the number of characters that are currently in the buffer (including end of line character if present). Ex see (5.26e).

Note4 For inputting in C (C++) see the following library functions from `<cstdio>`:
`scanf`, `gets`, `getchar`.

In principle, you have to distinguish two different ways of reading with `cin`:

- Reading with the **input operator „>>“**—as shown above:
 - At first, all possibly inputted white spaces are discarded; these are the characters regarded as white spaces: space, `horizontalTab`, CR ('carriage return'), LF ('line feed', new line), `newPage`, `verticalTab`.
 - Beginning of the actual reading, i. e. the transfer/conversion of characters into the variable belonging to them: with the first non white space character.
 - End of reading: by the first inappropriate character (when having the type string, this is a white space character);
important: this inappropriate character remains in the read buffer, so it is the first character to be read when performing the next read.
- **Using member functions**, see (5.24), (5.23b): no special treatment of whitespaces, each character is regarded as a character that has to be read.

- (5.22) Variable name for the **standard output device/unit**: `cout` (character out)
 Variable name for the **standard error output device/unit**:
`cerr` and C++(new) `clog` (character error, character log)
 Standard output: mostly screen, can be redirected on the operating system level.

Standard error output: mostly screen, cannot be redirected in DOS, in Unix/Windows NT a redirection is possible; `cerr` means an unbuffered output, `clog` a buffered one.

Statement for output: `{ cout | cerr | clog } { << Expression }_{1..n} ;`

- Note1 Really, more exactly `{ ... }_{0..n}`, but zero times normally senseless—like (5.21 Note1).
- Note2 In an output with `cout`, `cerr`, `clog`, the *Expression* may also be a manipulator, see (5.23c).
- Note3 Main effect of the above expressions: see (5.25c) (stream object as a reference).
- Note4 For outputting in `C (C++)` see the following library functions from `<stdio>`: `printf`, `puts`, `putchar`.

(5.23)

- (a) The variables `cin`, `cout`, `cerr`, `clog` from (5.21, 5.22) are so-called objects, i. e. variables of a specific type, a class type.

Objects can be handled by calling their **member functions**. The syntax for this:
ObjectName.MemberFunction(Parameter-LIST_{opt})

↗ Details about objects see (6.42a) and (9.13), about member functions (9.11b).

- (b) A member function which is especially important for error handling is `ignore()`; it misses (ignores) characters from the input stream; here, also white spaces (5.21 B_{ox}) are regarded as characters to be counted.

```
cin.ignore();
    exactly one character is missed (if not at end of file).
cin.ignore(number);
    number characters are ignored (if there is not end of file before).
cin.ignore(number, character);
    At most number characters are ignored, but maximum until the next appearance
    of character (included).
```

Ex (5.26d, 5.26e)

- (c) In addition to using member functions, you can use stream objects in expressions with the operators `<<` and `>>` (5.22, 5.21).

Within this kind of output, further forms for *Expression* (right hand operand in (5.22)) can be used, i. e. the so-called **manipulators**: these manipulate the data stream, but mostly do not produce any output, themselves. For most of these manipulators, there are equivalent member functions. All settings remain fixed until the next alteration; exception: a set of the minimum output width will be reset at once after the next output.

Note There are manipulators for the input, too; however, they are used more seldom.

The most important manipulators and the equivalent member functions:

Manipulator	Member function	Default	Meaning
<code>endl</code>	—		output new line and flush (i. e. empty) the output buffer
<code>flush</code>	<code>flush()</code>		flush the output buffer
<code>dec</code> , <code>oct</code> , <code>hex</code>	—	<code>dec</code>	set base for integer output
<code>setw(Width)[⊗]</code>	<code>width(Width)</code>	0	set minimum output width [∇]
<code>setfill(Character)[⊗]</code>	<code>fill(Character)</code>	'␣'	set fill character

[∇] will be reset to default value 0 automatically after next output
 —all other settings remain fixed until altered—

[⊗] include of `<iomanip>` necessary (always for manipulators with parameter(s))

For formatting the output of floating point numbers, there are the following possibilities (among others):

manipulator or M member function	meaning
<code>ios::scientific</code> <code>ios::fixed</code> M <code>setf(0,ios::floatfield)</code>	set output format for floating point numbers (default: format depending on value): scientific format (floating point notation with exponent), fixed point notation reset output format to default
<code>setprecision(IntegerValue)</code> [®]	(if <code>sci./fixed</code> ;) number of decimals after point (else;) number of leading digits, in total

[®] include of `<iomanip>` necessary (always for manipulators with parameter(s))

↑↑ Further overview see (C++/ch. 3.4)

(d)

Ex

```
#include <iostream> // Example E05-23.CPP
#include <iomanip>
using namespace std;
int main()
{
    // Only for counting the write positions of the output:
    cout << "1234567890123456789012345" << endl;

    int i=65;
    cout << setw(5) << i
         << setw(5) << setfill('*') << hex << i
         << setw(10) << 15
         << 15
         // reset: fill character, and base of integer representation
         << setfill(' ') << dec
         << endl;

    // The same manipulation with member functions:
    cout.width(5); cout << i;
    cout.width(5);
    cout.fill('*'); cout << hex << i;
    cout.width(10); cout << 15;
    cout << 15;
    // reset: fill character, and base of integer representation
    cout.fill(' '); cout << dec
    << endl;


    return 0;
}
```

For comprehending the above program, this is the output:

```
1234567890123456789012345
65***41*****ff
65***41*****ff
```

(5.24) If all characters, also white spaces (e. g. 'end of line character' '\n', cp. (5.21 Box)), are to be read as single characters, this can be done with `get()` as a member function.

Syntax, cp. (5.23a): `cin.get(Parameter-LISTopt)`

- (a) `cin.get(CharacterVariable)`: exactly one character is read into the *CharacterVariable*.
- (b) `cin.get(StringName, NumberCharacters)`: see (5.55c);  risky, better: `getline` (c)!
- (c) `cin.getline(StringName, NumberCharacters)`: see (5.55b).
- (d) In each of these three cases (a,b,c), the total expression `cin.get(...)` has two effects:
 - Side effect: character/string is read into the parameter.
 - Main effect: stream object (as a reference) with the state after reading, cp. (5.25c).

↑↑ More details see also (Cpp/ch. 4).

⚠ `cin.get()` (without parameters) is possible, too; the value of the character is returned. This should be assigned to a character variable. Pay attention if you read non-standard characters, e. g. the umlauts in German or other national special letters: surprising effects!! (Remedy: after (not before) the check on eof, use a type cast to `char`!) Better: do not use it! Instead of this: the above mentioned use of `get(CharacterVariable)` according to (a) does not produce any problems with umlauts or other special letters.

(5.25) Error handling with streams

- (a) There are three bits that characterize the internal stream error state, called `eofbit`, `failbit`, `badbit`, in addition also the pseudo bit `goodbit` which indicates that all error bits are clear:
- `eofbit` is set after a read attempt behind the end of the file (EOF)
 - in addition, the `failbit` is set, too –,
 - `failbit` is set after an error that can probably be repaired (e. g. read attempt after EOF, read attempt of a letter on `int`),
 - `badbit` is set after an error that causes the stream to be regarded as non-repairable (e. g. attempt to open a non existing file).
- (b) The test on the error state is possible e. g. with the member function `fail()`; it returns `true`, if `failbit` or `badbit` is set, otherwise `false`.
- More easy is the Boolean test of the stream object itself; it returns the inverted `fail()` value:
- `if (StreamObjectName) ...` is equiv. to `if (!StreamObjectName.fail()) ...`
 - `if (!StreamObjectName) ...` is equiv. to `if (StreamObjectName.fail()) ...`
- (c) Many expressions with a stream object as an operand produce the stream object itself as the main effect (more exactly: a reference on it (7.31)), i. e. with the state after the operation. Also these expressions can be tested as Boolean in order to get the error state of the stream object.

Ex1 Left: three expression statements; the main effect of each of the complete expressions before the semicolon will be discarded here (3.32a), it is the stream object itself in the state after the operation, details about `get()` see (5.24a);

Middle: error check with `fail` (TRUE if error);

Right: combination of the two parts with inverted Boolean test (therefore TRUE if error):

```
cin >> Variable;      if (cin.fail()) ...      if (!(cin >> Variable)) ...
cout << Expression;  if (cout.fail()) ...      if (!(cout << Expression)) ...
cin.get(Character);  if (cin.fail()) ...      if (!cin.get(Character)) ...
```

Ex2 Four variations, each with the same effect:

- (1) 'normal' program text: Read, then error test loop with the member function `fail()`,
- (2) similar, however not with `fail` function, but with Boolean inverted test of `cin`,
- (3) shortened (1) with comma operator (5.13), cp. (5.26d modification),
- (4) shortened (2).

The advantage of (3) and (4) is that the read statement does not appear twice (once before the loop, secondly as the last statement in the loop).

Regarding the error handling according to (d) mentioned in the comments: the error state must be cleared, and the error cause must be removed (not written in the example code, see e. g. (5.26d)).

```
int num;

/* 1 */ cin >> num;
while (cin.fail()) {
    // error handling according (d)
    cin >> num;
}


/* 2 */ cin >> num;
while (!cin) {
    // error handling according (d)
    cin >> num;
}
```


```


/* 3 */ while (cin >> num, cin.fail()) {
        // error handling according (d)
    }

/* 4 */ while (!(cin >> num)) {
        // error handling according (d)
    }

```

- (d)  In case of an error (one of the error bits set), the stream does not do anything any more—whatever you want it to do; therefore it is absolutely necessary (in the case of a repairable state): **clear the error bit(s)**. This is done by calling the member function `clear()`, cp. (5.26d).

 In addition, normally the error cause must be removed because otherwise the same error will be generated when repeating the operation. When reading e. g., the character that caused the error remains in the stream! So at least one character must be discarded, e. g. by `ignore` (5.23b). Mostly it is sensible to discard the complete input line, cp. (5.26d).


 *Additional information about finding and handling errors with streams see (C++/ch. 4).*

- (e) The member function for testing the **eof condition** (i. e. if `eofbit` is set) is the `eof` Funktion:

StreamobjectName.eof()

It returns **true**, if `eofbit` is set (i. e. after a read attempt behind the end of a file), else **false**. This function is important when handling files (see next subchapter), especially if you want to distinguish a general error from the special eof error.


- (f) If the read operation into a variable fails, its old value should not be changed. This is also true with the predefined stream objects, according to (Str3/ch. 21.3.3). Unfortunately, the compiler Microsoft Visual C++ 6.0 does not obey this rule, it clears the variable with the zero value in the case of error.

 In the following program fragment, the number 35 should be outputted in the case of an error; when compiled with MS VC++ 6.0, the value 0 will be written, however.

```

int num=35;
cout << num << endl;           // 35
if (!(cin >> num)) {
    // if read error:
    cout << num << endl;       // should be value 35!
                                // MS VC++ 6.0 however: value 0
}

```

(5.26) 

- (a) Bad copy program (why??):

```

#include <iostream>           // Example E05-261.CPP
using namespace std;
int main()
{
    char c;
    while (cin >> c) cout << c;
    return 0;
}

```

- (b) Correct copy program:

```

#include <iostream>           // Example E05-262.CPP
using namespace std;
int main()
{
    char c;
    while (cin.get(c)) cout << c;
    // possible also symmetrically:
    // while (cin.get(c)) cout.put(c);
    return 0;
}

```

- (c) Filter program:

```
#include <iostream>      // Example E05-263.CPP
using namespace std;
int main()
{
    char c;
    while (cin.get(c)) {
        // manipulation (see lecture)
        cout << c;
    }
    return 0;
}
```

- (d) Program fragment: catching incorrect user inputs, i. e. input of non-digit characters when reading into an integer number, cp. (5.25):

```
#include <iostream>      // Example E05-264.CPP
using namespace std;
// ...
int number;
while (!(cin >> number)) {
    cin.clear();           // absolutely necessary!
    cin.ignore(1000, '\n'); // empty the line in OS buffer;
                          // another possibility see (e)
    cerr << "Please, input digits!" << endl;
}
```

Modification: the *while-statement* with use of the comma operator (5.13)—possibly clearer (?), discussion in lecture:

```
while (cin >> number, !cin) ...
```

- (e) Program fragment: Emptying the input buffer
- `cin.get()`
- , cp. (5.21 Note3); remarks on the variable definition see (4.51b)

```
#include <iostream>      // Example E05-265.CPP
using namespace std;
// ...
if (int num=cin.rdbuf()->in_avail())
    cin.ignore(num);
// ...
```

Alteration for `cin.get()` see lecture.

5.3 Handling text files

- (5.30) `cin` If you have understood how to handle standard stream objects (see last subchapter), you will notice that dealing with text files is very easy in C++. Using files with the help of stream objects is done just exactly as the use of `cout` and `cin`. What is new about it is that you have to do something additional before using the objects, and to do something after use: before use, generate a stream object and link it to a file (5.33); after use, you should remove this link (5.34). The example program copying a file (5.35) demonstrates all these steps.

Dealing with binary files will be covered later on (ch. 12.6).

- (5.31) You must distinguish between two kinds of files (or two ways of handling a file):

- **Text files:** they render normal text, they are organized in lines, special format control characters are not allowed—except the white spaces (5.21 B_{0x}) and (possibly) an end of file character (ASCII dec. 26, cp. (5.21)).

Here, the C++ run time system is responsible for the possibly necessary transformation between the operating system’s internal representation of a new line code and the C++ representation ‘\n’.

Ex In Unix, no transformation would normally be necessary; in DOS/Windows the new line is represented by the double characters CR/LF (ASCII dec. 13/10, see (1.45a)), in C++ by ‘\n’ (mostly as character LF, ASCII dec. 10).

- **Binary files:** files with arbitrary bytes; these bytes are not interpreted in a special way during handling, they are transferred unaltered between the operating system and the C++ program.

Here only dealing with text files is explained. Handling binary files is discussed in (ch. 12.6).

(5.32) Handling text files

In C++, dealing with text files is very easy, it is similar to the use of the standard input `cin` and the standard output `cout`.

- (a) No alteration is necessary if there is a redirection of standard input or standard output on the operating system level: normal use of `cin` and `cout`.
- (b) In contrast to (a), if you deal directly with files, you have to consider the following steps:
 - Preparations, see (5.33):
 - Creation of a stream object of appropriate type by variable definition,
 - link this variable to the wanted file: **open** the file.
 Both steps can also be performed with one single statement.
 - Work with this file like standard input or output: read and write (see the previous subchapter). Instead of the names `cin`, `cout` you have to take the stream object name (variable name).
 - End activity, see (5.34): **close** the file and destroy the stream object.

(5.33) Preparation of stream objects

- (a) Type of needed stream object (class name):

`ifstream` if you intend to read,
`ofstream` if you intend to write,
`fstream` for both read and write.

In order to make the names available, you have to include the appropriate header file

```
#include <fstream>
```

The needed stream object is created e. g. by the following variable definition:

```
StreamObjectType StreamObjectName ;
```

- (b) Linking this stream object to the wanted file is done by invoking the `open()` member function:

```
StreamObjectName.open(FileName);  
StreamObjectName.open(FileName, Mode);
```

You have to write the *FileName* in the operating system’s specific format, additionally, if you need, including a path.

⚠ For a path in DOS/Windows if cited as string constant, you have to take the double characters ‘\’ instead of the single character ‘\’, cp. (3.25)—in contrast to (5.74)!

You can use the following constants for the parameter *Mode*:

```
ios::in      read mode (default for ifstream)  
ios::out     write mode (default for ofstream)  
ios::app     append when writing  
ios::ate     positioning to the end of the file ('at end')  
ios::trunc   delete previous content (truncate)  
ios::binary  binary access, see (5.31, 5.36); without this: text mode
```

These single modes can be combined with the operator ‘|’ `Op12` (5.11), e. g..

```
ios::in | ios::out.
```

With previous compilers—not necessarily the very new ones—, additionally the following flags are available: `ios::nocreate` and `ios::noreplace`

If you try to open a non-existing file for reading (with no *Mode*):

`C++(new)` error message, file is not created,

`C++(old)` file is created as an empty file; the mode `ios::nocreate` prevents it.

Rec *The success or failure of opening the file should be checked as the stream state (5.25b) before any further operation.*

- (c) Both steps (a) and (b) can be combined into a single statement (possibly not with `fstream`):
- ```
StreamobjectType StreamobjectName(FileName);
StreamobjectType StreamobjectName(FileName, Mode);
```

- (5.34) Closing the stream is done by invoking
- ```
StreamobjectName.close();
```

Destroying the object is done as with destroying other variables, e. g. by leaving the block in which it was created.

If the stream is not closed before, this is done implicitly by destroying the object (destructor call (9.23a)). However:

Rec *Get used to an explicit close by the member function `close()`.*

- (5.35) **Ex** Copy a text file—not appropriate for binary files!

```
#include <iostream>           // Example E05-35.CPP
#include <fstream>
using namespace std;

int main()
{
    ifstream in;
    in.open("FILE.TXT");
        // Mode "ios::in" not necessary because of type ifstream
    // Or only with one statement:
    // ifstream in("FILE.TXT");

    if (!in) {
        cerr << "File FILE.TXT not found or not allowed to open"
              << endl;
        return 1; // Value 1 as error code
    }

    ofstream out("NEWFILE.TXT");
        // Mode "ios::out" not necessary because of type ofstream
    // Or with two statements:
    // ofstream out; out.open(...);

    if (!out) {
        cerr << "File NEWFILE.TXT cannot be created!" << endl;
        in.close(); // can be missed
        return 1;
    }

    char c;
    while (in.get(c)) out.put(c);
    // Or also:
    // while (in.get(c)) out << c;

    in.close(); // can be missed
    out.close(); // the same
    return 0;
}
```

- (5.36) Only parts of the above explanations can be applied when dealing with binary files.



If you have binary files, you must set the mode `ios::binary` (5.33b); reading and writing is best performed with the member functions `read` and `write`, see (ch. 12.6) and (C++/ch. 4).

5.4 Arrays

(5.40) **Array**: putting together several elements of the same type, all together as a single type.

Mathematics:

- one-dimensional: vector,
- two-dimensional: matrix.

In order to avoid confusion, here this construct is deliberately not called vector (as, in contrast, in the most C++ and C books) because vector as mathematical term has a different meaning.

The array is very often used in programs for natural scientific and technical applications. Since some authors presumably have their focus on other application fields, this topic is explained only very briefly in some textbooks. You may avoid the problems made by the extremely serious error of an index overflow (5.41 Δ) by moving to types of the new C++ run time library (not discussed in this course). But in scientific-technical applications, this is not always sensible and—with precise (reflective) programming—not necessary either.

(5.41) *Declaration*^{10/11} of an array type (one-dimensional) \doteq

Type *ArrayName* [*ConstExpression*] ;

ArrayElement (as a variable) \doteq *ArrayName* [*Expression*]

- The aforementioned *Type* is the type of the single elements of the array.
- *ConstExpression* sets the number of elements of the array (it must be calculable at compile time).
- The numbering of the elements ranges from 0 up to *ConstExpression*-1.
- Δ An index range overflow of *ArrayElement* is not checked!

```
Ex int vector[20];           // definition: array with 20 int elements
    vector[0]=-12;         // assignment to the first element
    vector[19]=23;         // assignment to the last(!) element
    // now all elements of the array shall be set; output of all elements according to their order:
    int index;
    for (index=0; index<20; ++index) // sensible: "index<number" instead of "index<=number-1"
        cout << setw(5) << vector[index]; // setw() see (5.23c)
```

(5.42) Two-dimensional array: take double indices.

Declaration of a general array (one-dimensional or multidimensional):

^{10/11part} *Declaration* \doteq

^{12/14} *Type* [^{30/31} *ArrayName* [[*ConstExpression*]]_{1..n}]⁻¹ *LIST* ;

Use of the array element: *ArrayName* [[*Expression*]]_{1..n}

```
Ex double matrix[20][15]; // definition: matrix with 20*15 double elements
    matrix[3][12]=-2.3;   // assignment
    // assign the value -1.6 to all elements:
    int row, column;     // meaning of row/column (see assignment below) according to matrix in maths
    for (row=0; row<20; ++row) // loop with "rows"
        for (column=0; column<15; ++column) // nested loop with "columns"
            matrix[row][column]=-1.6; // assignment to each element
```

(5.43) Arrays may be initialized:

- one dimensional: with a list of values enclosed in a pair of brace brackets (strings also with a string constant, see next subchapter),
- multi dimensional: with nested lists, a pair of braces for each nesting level.

If there are less values than needed, the rest will be filled with zeros or zero values; too many values generate a compile time error.

```
Ex int arr[20] = { 2, 5, 7, -23 }; // Note: the remaining 16 values are filled with 0.
    double matrix[3][2] = { { 1.0, 7.3 }, { -0.7, 6. }, { 3.7, 5.3 } };
```

When initializing an array, it is permissible to leave the (innermost) dimension because—in this case—the compiler itself counts the number of elements, see (5.51, 12.71c).

- (5.44) In particular, symbolic constants (4.72) should be used as array bound because they make a program more maintainable; therefore, the bounds can be changed more easily later on.

Ex If there is an array bound of 50, and this number shall be altered, each occurrence of 50 and of 49 (as last permissible index) must be checked; an automatic replacement is not possible because these numbers can also have another meaning. However, an alteration of the value of the constant `maxNum` (and thus implicitly of the value `maxNum-1`) can be easily performed in its definition.

5.5 Strings (C strings)

- (5.50) **Ovw** Strings may be encountered very often in programming practice. In this course, strings are introduced only in the (older) form of so-called C strings. Indeed, these strings' handling is more difficult because they are arrays of characters and thus present the great risk of memory overflow. But when you are used to handle arrays in general (see last subchapter), the C strings will not cause difficulties.

These special character arrays have an additional property exceeding general array properties: the null character termination (5.51). As with all arrays, a copy by assignment is not possible (5.52). The most important string functions are introduced in (5.53). The specialities of input and output are covered in (5.55).

Note In `C++(new)` there is also a library class `string`; objects of this class may be handled more easily, peculiarly memory overflow is hardly possible. As mentioned above, this class is not explained in this course, however.

- (5.51) **String** (more exactly: C string) (String, Zeichenkette):
special array of `char` with setting the actual length: behind the last actually used character, there is the character `'\0'` (null character, i. e. all bits 0—not the digit `'0'` whose value in ASCII is dec. 48 (1.45a)).
Thus, the minimum storage requirement of a string is 1 greater than the number of used bytes.

Example:

```
char line[81];
char message[20]="Hello!";
```

The `char` array `line` is not initialized, it must be set sensibly later on. The array `message` is initialized (no assignment, but settling the first value, cp. (5.52)), closing null character is automatically added.

⚠ Dimensioning must be at least `NumberCharacters+1!!`

↑↑ When initializing in `C`, also `NumberCharacters` as dimension is allowed, but this is very dangerous **⚠** because the null character is missing. In `C++`, this error possibility is not allowed.

You can leave the compiler to count the needed number, but however only when initializing; the compiler automatically sets the correct number including the null character:

```
char newMessage[]="Hello!";
```

- (5.52) Copying one string onto another is not possible by assignment, but only by copying all characters including the null character, see also (5.53a). This is generally valid for arrays, cp. (5.65b). Initializing (with constant values, not with variables), however, is allowed (5.51).
- (5.53) Four important string functions (necessary for using them: `#include <cstring>`):

- (a) `strcpy(TargetArray, SourceArray);`

The function copies onto `TargetArray`, i. e. all characters of `SourceArray` until including the (first) null character, so the actual source string is copied.

⚠ A check whether `TargetArray` has enough memory, cannot take place! Responsibility of the programmer! Important source of error!

- (b) `strcat(TargetArray, SourceArray);`

Appends (concatenates) the *SourceArray* to the end of the *TargetArray*, i. e. the previous null character of *TargetArray* is overwritten by the first character of *SourceArray* etc. Also here, there is no check on the needed memory!

(c) `strlen(CharacterArray)`

returns the actual string length, i. e. the number of used characters without the terminating null character; this length must be calculated within the function.

⚠ *The returned type is an unsigned type, therefore, pay attention if mixing with (normal) signed types, cp. (12.23 ⚠)! If the value is needed in an expression: first assign this value to an int variable, and use this variable afterwards.*

(d) `strcmp(Array1, Array2);`

compares both strings, indicates the alphabetical order of them (according to the internal character representation):

- return value < 0, if *Array1* < *Array2*, i. e. alphabetic order means *Array1* before *Array2*,
- return value == 0, if *Array1* equal *Array2*, i. e. *Array1* and *Array2* are equal,
- return value > 0, if *Array1* > *Array2*, i. e. alphabetic order means *Array1* after *Array2*.

(5.54) Empty string, null string: string of the length zero, i. e. with the null character as the first character; empty string as a constant: `""` (two double quotes without a space between them).

(5.55) Supplements to (ch. 5.2):

`arr` shall be a valid string variable.

(a) `cin >> arr` reads exactly one 'word': at first ignoring all white spaces, then reading the character sequence, finishing with the first inappropriate character, i. e. here with a white space; this white space character remains in the read buffer.

There is no check on memory overflow of `arr`.

⏪ *Limiting the number of characters to be read by the operator >> is possible e. g. by `cin.width(NumberCharacters)`. Hereby—however only during the next reading—at maximum `NumberCharacters-1` will be read. Rec better (b,c).*

(b) `cin.getline(arr, NumberCharacters)` reads exactly one complete line, but maximum `NumberCharacters-1` characters, additionally generating the null character termination in `arr`. With correct size, there will be no memory range overflow. Recommendation: at best, take the correct `NumberCharacters` by `sizeof(arr)` (Op3j), see (5.14).

The new line character `'\n'`, however, is read (if reading is not finished before), but it is not copied into the string.

(c) Only for special applications, see ⚠!

`cin.get(arr, NumberCharacters)` works like `getline`; but the new line character remains in the read buffer. In this way, you can check afterwards, if you read a complete line.

⚠ *When using this function multiple times one after the other without any read instructions in between: the reading will not continue! Risk of an infinite loop! So keep in mind: the normal function should be `getline` (b).*

(d) Array name `arr` as a formal function parameter:

⚠ *Within the function, the expression `sizeof(arr)` is interpreted very differently; instead of this, you should use `sizeof(ArrayType)`, see also (7.46 ⏪).*

(e) `cout << arr` outputs the complete string (including white spaces possibly present in `arr`).

5.6 Overview of types

(5.60) Ow A **type** is a name or a description for a set of values including the allowed operations belonging to them, cp. (2.25a2), (3.20).

Here you read how a user (programmer) may generate new types (type names) and use them (5.61, 5.62).

Frequently an expression of one type must be converted into another type. This can be done implicitly (5.63), i. e. without extra denotation by the programmer. But an explicit type conversion is possible, too. Type conversions (“type casts”) are a frequent source of errors in the language C if the programmer does not have sufficient knowledge (or—if yes—he or she does not apply it when programming). Therefore C++(new) has introduced less error prone type cast operators (especially `static_cast`, see (5.64a) and remark (5.64↑12)). These new operators are preferable to the older C conversions (5.64c) and to the also error prone conversion of old C++ (5.64b), although these new operators may hardly be exceeded regarding the ponderousness of spelling (extremely long names!).

In (5.65) the pointer type is mentioned. At first, only its basic meaning is important because the term ‘pointer’ is needed for some explanations. The most important explanation is already indicated in (5.65b) (automatic conversion of an array name into a pointer). The consequences will be mentioned soon (7.46). A direct application of the pointer type will be given much later, to be precise, from (ch. 10.2) onwards.

- (5.61) In many programming languages—also in C++—there are types that are known by the compiler, the so-called **built-in types**.

Ex `int, float, unsigned char.`

Additionally, users (programmers) are allowed to build their own types: **user-defined types**. These types (with their inner structure) must be introduced to the compiler before they can be used. When using simply built user-defined types, the type definition is often combined with a variable definition.

Ex `int arr[34];`
A variable with the name `arr` is created, its type is ‘array with 34 `int` elements’.

When using a more complicated type, it is sensible (sometimes necessary) to introduce the type explicitly to the compiler before using it: **type definition**. This can be done with a **typedef** (5.62), or without a **typedef** if using classes or similar (9.11).

- (5.62) Definition of a type with **typedef** in C++/C:
Introducing a new name, synonymous to the given type; no introduction of a new (unique) type.

TypeDefinition \doteq `typedef Type Name ;`

Without the keyword **typedef**, *Name* would be a variable name; with the preceding **typedef** this name is made to a type name for the given type.

Ex Synonyms in the following example: types `int` and `NoFraction`, further types `Line` and ‘char array with `max(=81)` elements’.

```
// ...
typedef int NoFraction;
NoFraction newNumber; newNumber=32; cout << newNumber;
const int max=81;
typedef char Line[max];
Line inputLine;
cin.getline(inputLine,max);
// or (better): cin.getline(inputLine,sizeof(Line));
// ...
```

- (5.63) **Implicit type conversions** are normally appropriate if without compiler warnings, i. e. conversion with unaltered value from `char` into `int`, from `int` into `double`, from `float` into `double` (less precise type into a more precise type).


Riskier are conversions if some information gets lost, e. g. by assignment of a `double` value to an `int` (fractional part is cut, perhaps there is a range overflow) or assignment of an `int` value to a `char` (risk of range overflow). Mostly, the compiler outputs warning messages; if applicable, an explicit type conversion should be used (5.64) to avoid these warnings.

- (5.64) **Explicit type conversion or type cast**


- (a) **C++(new)** New operator (Op2h):
`static_cast<Type>(Expression)`

With this, a conversion of *Expression* into *Type* is made, i. e. with check already at compile time whether such a conversion is sensible.


`Ex` `int i; double d=34.9; i=static_cast<int>(d);`

- (b) `C++`  `Op2c`; with this, also 'absurd' conversions can be made, therefore more risk! You use the new type as a function name; so this is allowed only if the new type is a name.

`Ex` `int i; double d=34.9; i=int(d);`


- (c) `C/C++`  `Op3i`, effect as `Op2c`, but, however, not so clear; here no *TypeName* like `Op2c` necessary.


`Ex` `int i; double d=34.9; i=(int)d;`

 *Because of risk and confusion, both of the latter conversions (b,c) are now regarded as deprecated (Str3/ch. B.2.3).*

There are many conversions which are (at least for the beginner) very dangerous because he or she cannot survey the consequences. Therefore, obey the recommendation in the following box, absolutely!


When getting compiler warnings about type conversions, firstly you should think about whether the compiler regards it with the same meaning as you. If yes, use `static_cast` (a); after this, the warning should not appear any more.

 *More details about type conversions, especially implicit ones, see (Cpp/ch. 7).*

 *Additional type conversion operators `Op2i-k` see C++ books, in (10.24 Note4) and (12.63a, 12.64a) you find some remarks about the `reinterpret_cast`.*

(5.65) Data type **pointer** <Zeiger>

- (a) The data type pointer is characterized by interpreting its value as the address of another variable, the pointer 'points' to another memory location.

 *More details, especially the definition und use of the operators belonging to them, see (ch. 10.2).*

- (b) IMPORTANT, however, in this context:

An array name is automatically converted by the compiler into a pointer to the first array element (element with the index 0). There are surprising consequences in functions with array parameter, see (7.46); this, also, is the reason why arrays cannot be copied by assignment (5.52).

5.7 Preprocessor

- (5.70) `Ow` The preprocessor is a text processor, which adapts the source text before the compiler gets it. It works line oriented, independently of C++/C syntax. In older programs, especially in C, preprocessor properties are used very frequently. In C++(new) this is not necessary in most cases because better constructs are available.

The `#include` (5.74) is still often used, furthermore the conditional compilation (5.75): each header file (8.24) made by you must have an include guard (5.75c) which prevents multiple inclusion. The reason why this is necessary will be explained later, especially from (ch. 9) onwards.

Syntax for line with a preprocessor directive:
begin of line with `#` (after optional white spaces).

- (5.71) **Line continuation character** (for a preprocessor line, unnecessary for compiler lines): `\` with an immediately subsequent 'new line character' (no space before); the preprocessor discards the `\` and the new line character, with this it combines the two lines into one line.

(5.72) Macro without parameter

`C (C++)` very often used for symbolic constants; `C++` better: `const ...`

`C/C++` used for definitions in conditional compilation, see (5.75) .

(a) #define Name TextReplacement

Semantics: beginning with the following line, *Name* (as a lexical part, not within a string) is replaced by *TextReplacement* ('silly' replacement, independent of syntax); *TextReplacement* is allowed to contain spaces, too. If *TextReplacement* is missing, *Name* is deleted, i. e. replaced by nothing.

`Ex`


```
// Replacement for recommendation in ch. 0.3
#define bool int
#define true 1
#define false 0
```

(b) The replacements are finished at the source text end or at appearance of:

`#undef Name`

After this, *Name* not known any more for replacement.

An `#undef` with an unknown name is harmless, a `#define` with a known name produces an error, in general.

(5.73) Macro with parameter  (10.12b)

`C++` better: `inline ...` (10.12a) – is not yet discussed here.

(5.74) Inclusions

```
// Form 1:
#include <FileName>
// Form 2:
#include "FileName"
```

Includes the file *FileName* (in general, a so-called header file) at this location. This directive is nestable.

- Form 1: Searches at the locations known to the preprocessor (generally, directories for general header files)
- Form 2: Searches at first in the actual (default) directory, afterwards as form 1; this form is applicable for user header files. If *FileName* includes a path (DOS/Windows: character '\ ' not twice in contrast to (3.25) and (5.33b)), only the given path is searched through.

⚠ Unfortunately, spaces within `< >` and `" "` are significant, therefore pay attention!

(5.75) Conditional compilation

```
(a) #if ConstExpression1
      TextLines1
    #elif ConstExpression2
      TextLines2
    #else
      TextLines3
    #endif
```

Semantics:

- If *ConstExpression1* is TRUE (i. e. unequal 0), only *TextLines1* are taken, *TextLines2* and *TextLines3* are ignored.
- Else: if the `#elif` line exists and *ConstExpression2* is TRUE (unequal 0), only *TextLines2* are taken, *TextLines3* (and of course also *TextLines1*) are ignored.
- Else: if the `#else` line exists, only *TextLines3* are taken.
- Else: no text lines between `#if` and `#endif` are taken.

- (b) Instead of the `#if` line in (a), you can take one of the following lines, too:

```
#ifndef Name
#endif Name
```

Semantics: TRUE if *Name* (by a `#define`) is defined (`#ifdef`) or is not defined (`#ifndef`), an empty definition without *TextReplacement* is regarded as a definition, too. Continuing the preprocessor lines (with `#endif`, possibly also with `#else`) see (a).

- (c) Application e. g. for prevention of multiple inclusion of a header file ('include guard'). Here you can choose the name arbitrarily (in the example it is the name 'EXAMPLE.H_'), but it has to be unique within the range of all program files and must not conflict with any C++ names. Therefore, a name derived from the filename is sensible.

```
// Example header file EXAMPLE.H
#ifndef EXAMPLE_H_
#define EXAMPLE_H_
// Definitions of this header file
// ...
#endif
```

A double `#include` includes the definitions of the header file only once:

```
#include "EXAMPLE.H"
#include "EXAMPLE.H" // harmless even if double definitions not allowed
```

Further application: commenting out larger text parts (also with included comments), nestable, cp. exercises:

```
#if 0
    TextLines—are ignored
#endif
```

(5.76) Further activities of the preprocessor (e. g.)

- String constants separated by white spaces (new line, too) are concatenated; so very long string constants over several lines are possible.

```
Ex cout << "This is a very very "
    "long string" << endl;
```

- A comment is replaced by a space.

6 Creating construct units, problem of distinction between hidden and accessible (language-independent reflection)

6.0 Overview

This chapter—just as ch. 2 independent of a special programming language—renders the basic principles of how to deal with larger programs to preserve clarity.

Subch. 1 shows that this clarity for humans is fundamentally possible only by dividing the problem area into parts. These parts—*independent of the way they are built*—are called **construct units** here. This term is *not* a fixed term in computer science; it is chosen deliberately to describe the basic properties of the divided parts independently of their realization according to a chosen programming style. Without understanding the principle of how to build and to use construct units, no software development may be done. A sensible design of such construct units exploits the opportunity that the outer view and the inner view of these construct units can be uncoupled from one another to a large extent by encapsulation. The outer view answers the question of ‘What?’, the inner view the question of ‘How?’: *What* does the construct unit achieve? *How* does it realize this?

After this, you will learn how to build and to use such construct units in three different programming styles.

- In **procedural programming** (subch. 2), so-called functions or procedures are built each containing a small or also a larger part of the solution path, i. e. encapsulation of a series of actions.
- **Modular programming** (subch. 3) encapsulates several related functions together with the data belonging to them. These data exist exactly once per module.
- **Object oriented programming** (subch. 4) also generates capsules consisting of functions and data, the structure of which is settled in the form of so-called classes. At run time, you may create as many objects as wanted from these classes (construction plans) which may be used and destroyed independently of each other.

Additionally, in subch. 3 the term **storage class** will be introduced to characterize how and for how long memory is reserved for data at run time. Two different storage classes will be presented: ‘automatic’ (limited life time at run time with automatic administration by the run time system) and ‘static’ (memory space reserved during the complete program run time).

6.1 Design of systems: construct units and hiding principle

- (6.10) Ovw Larger programs may be controlled by humans only when the problem area is divided into parts; the way of division strongly depends on the programming style (6.11). These parts are called **construct units**, here.

Completely independent of the programming style, it is reasonable (actually even imperative) to build these construct units in an encapsulated manner; in this way, it is possible that the two views of them (namely the view **from outside** and **from inside**) are decoupled from each other to a large extent. The first view answers the question ‘What?’ (i. e. What does the construct unit do?), the second one answers ‘How?’ (i. e. How does it realize this?), see (6.12). With this **extensive decoupling** of the outer and inner view (6.13), using the construct unit and building its internal structure are independent of each other; so this may be done e. g. by different persons or groups. For use (or application), only the knowledge of the What is necessary, for construction you need the answer to the How.

- (6.11) **System design**

According to nature of humankind, a person can grasp and describe only a limited part of the world at one time. Thus, there is only one possibility to handle complex systems: to

divide the system logically (and physically, too). Afterwards, one can put the parts together to obtain the complete system.

There are different programming styles depending on the criterion how the whole is to be divided. This criterion can be subject to a ‘Weltanschauung’ (world view), namely a kind of view of how to observe and grasp the world. Two kinds of them are selected here because they are important for this course:

- **Flow oriented (sequence oriented) decomposition, ‘structured programming’** (also algorithmic or procedural decomposition):
The decomposition (dividing, breaking down) is done according to the sequence of actions (strictly chronologically).
- **Object oriented decomposition, ‘object oriented programming’:**
The decomposition is carried out according to the units which are found to have a certain independence in the world to be modelled (‘objects’ like things, humans, events, or made-up units, e. g. organizational units). These objects are given a certain amount of responsibility for themselves; they communicate with each other by exchanging messages and, by doing this, asking other objects to perform some partial tasks that they themselves are allowed to delegate.

Furthermore, there are also the data oriented decomposition, as well as different combinations.

(6.12) In each kind of decomposition—*independent of the decomposition criterion (6.11)*—, it is appropriate to build **construct units** which are sealed off from others. Frequently, these units contain construct units of still finer granularity, or are contained in ones of larger granularity.

The construct units (plug in components, procedures/functions, modules, objects) can be regarded—if properly designed—in two aspects independent of each other to a large extent:

- **From outside: ‘What?’**
the **user** of a construct unit must know what a unit does, not how it does it (‘black box’). Furthermore, he or she can use this unit arbitrarily often without constructing it once more. Necessary for the proper application: exact knowledge of the transfer facility (interface).
- **From inside: ‘How?’**
the **architect** or builder of the construct unit must know how to realize the demands the user expects, e. g. which algorithm or data to use. He or she does not need to know the use cases as long as the interface agreement is followed.

Note *In accordance with both of these viewpoints, we will distinguish between the architect of a construct unit and the user of this unit (a programmer with not necessarily less programming skill).*

Ex In many programming languages, the sine function is implemented:

- From outside: the user of $\sin(x)$ only has to know that x has to be measured in radian.
- From inside: the architect must implement an appropriate approximate polynomial for calculating the sine.

In order to achieve correct interaction between the outside and inside viewpoints (between user and architect), it is necessary to exactly determine the **interface** (Schnittstelle, Interface), i. e. the transfer facility between outside and inside, between user and architect views.

(6.13)

(a) It is a great advantage if the construct unit’s internals are hidden from the user to a large extent (**hiding principle**), and if he or she is able to influence only a small part directly (**public**):

- By doing this, user and architect can be different persons or even different groups who work more or less independently of each other.
- The internals can be altered later on without changing the many places where the unit is used. For example, later on perhaps you recognize that the internals—now—are not good enough or do not react properly in some (at first unimportant) special cases.

Note *The Y2K ('year two thousand') problem is a good example. On the other hand, the programmers of the sixties and seventies did not suspect that their programs would be used until the year 2000. Furthermore, they had not yet made the distinction between hidden and accessible, the programming languages hardly supported it. Additionally, memory for temporary and permanent storage was very expensive.*

(b) The protection of the internals from the user is of variable quality, depending on the programming language and the skill of the architect:

(1) Such protection is good which the user does not evade by absent-mindedness or carelessness.

Note *Also programmer-user are humans who do not always think about everything!*

(2) A better protection could be that which also prevents an intended invasion into the internals by the user, however noble the purpose: e. g. 'tricks' during programming so that the code (in this special case) is executed faster.

Note *'Tricks' are honorable for the programmer, at first; but afterwards there are large disadvantages namely if another programmer (or he/she himself after some time) wants to eliminate bugs or to extend some facilities.*

Nowadays, the hardware is normally so fast that a faster execution is less important than a good maintainability.

(c) With respect to a later software's maintainability (alteration, debug), additionally with respect to the ability to be extended and reused in other projects, the following **rules** apply:

- **hiding:** as many details as possible should be hidden from the user ('information hiding').
- **public:** only the absolutely necessary details should be accessible to the user, to be precise via the agreed interface (in general).

(6.14) The consequences of (6.12, 6.13) are very different. They depend

- on the use case,
- on the architect's 'Weltanschauung' (world view) (6.11),
- on the programmer's far-sightedness,
- on the programming language used,
- on the skill with which the programming language is used.

Three important programming styles and their construct units will be discussed in more details:

- procedural programming (ch. 6.2),
- modular programming (ch. 6.3),
- object oriented programming (ch. 6.4).

6.2 Procedural programming

(6.20) **Ov** In procedural programming, each construct unit consists of a set of actions. Therefore these units describe the activity in order to execute a small or larger part of the problem's solution.

Depending on whether the set of actions returns a value to the user or not, we distinguish two different kinds of these construct units: functions and procedures (6.21). The point (6.22) describes how the pseudo code represents the construction of the functions or procedures (named 'definition') and their use (named 'call' or 'invokation').

The interface between the outer and the inner view is realized in the form of parameters (6.23) (in addition to the possibility of returning a value). The parameters of a function call (i. e. the function's use) are called 'actual parameters', the ones in the function's definition 'formal parameters'. Additionally, it is very important to comprehend the two parameter passing modes presented here, and to be able to apply them: 'call by value' (copy the actual parameter's value onto the formal parameter) and 'call by reference' (the formal parameter is made to a synonym of the actual one).

Point (6.24) renders an example for the (nested) dissection of a problem into procedures and functions.

(6.21)

- (a) If you apply flow oriented decomposition (6.11)—this can be appropriate with smaller problems—, often it is resonable to encapsulate parts of the activities and to name them in order to be able to use them many times. Such construct units are called functions, procedures, subroutines—depending on the programming language.
- (b) According to most programming languages, we want to distinguish **two different kinds** of these construct units:
- **Function:** the unit returns—after performing its task— a value that can be used by the user, e. g. as is usual in the mathematics’ functions (the function value). This value is usually called ‘return value’.
 - **Procedure:** the unit—after performing its task—does not return any value to the user interface, e. g. a print task. Thus, there is no return value.

The names for these two kinds are different according to the programming language, e. g.:

- Pascal (names as used above): function, procedure (Funktion, Prozedur),
- Fortran: function, subroutine (Funktion, Unterprogramm),
- C++ and C: function (Funktion) for both.

In our language-independent reflections, we take the terms ‘function’ and ‘procedure’; the generic term is to be ‘function’.

- (c) The sense of the hiding principle in this case is that the only interface, i. e. the only facility for communicating between inside and outside (6.12), is the parameter list, additionally if existing the function’s return value; details see (6.22, 6.23).
- (d) The programming style that essentially uses only such construct units is called **procedural programming**.

Excellent examples for applying this programming style are calculations from input values, e. g. $\sin(x)$. The calculation algorithm is absolutely separated from its use.

(6.22) Agreement on how to write procedures and functions in pseudo code:

- (a) Construction plan of the unit, mostly called **definition** (Definition):

```
PROCEDURE newProc(formalPar1 TYPE type1, formalPar2 TYPE type2)
  // ...
  // optional:
  IF ... THEN
    RETURN
  // ...
  // optional:
  RETURN
END PROCEDURE
```

```
FUNCTION otherFunc(formalPar1 TYPE type1, formalPar2 TYPE type2) TYPE typeFunc
  // ...
  // optional:
  IF ... THEN
    RETURN VALUE ...
  // ...
  RETURN VALUE ...
END FUNCTION
```

Both construct units are allowed to have so-called formal parameters by which you can communicate with their interior; above there are two parameters, each. Details see (6.23).

Both units use a new keyword RETURN; it means that the control flow leaves the unit at this location.

In a procedure (directly before END PROCEDURE), you are allowed to miss RETURN because the procedure is left there anyway.

In a function, in connection with the RETURN statement, you must set the value that is to be returned; thus, you are not allowed to miss the value at end (directly before END FUNCTION) because otherwise there would be no return value. Additionally, the function

name—not the procedure name—has a type, namely the return type, in the above example ‘typeFunc’.

- (b) The unit’s use or its application is named a **call** (Aufruf) (verb: to call or invoke a procedure/function); it is carried out using its name, additionally if applicable the so-called actual parameters belonging to it.

```
newProc(actPar1, actPar2)
otherFunc(actPar1,actPar2)
```

```
BEGIN
  VARIABLE x, y TYPE Integer
  x ← 23
  newProc(16, 2*x)
  y ← otherFunc(x, 34) + 3
  Write x, y
END
```

(6.23) Parameters

- (a) You must strictly distinguish between the following two kinds of parameters:
- **formal parameters:** they are introduced at the definition, they are forms, shapes, covers, moulds, still without content.
 - **actual parameters:** they are used at the invocation, they are the contents belonging to the covers of the formal parameters.

Each actual parameter (at the call’s location) corresponds to exactly one formal parameter (in the definition); number and order, furthermore the type (perhaps after an allowed conversion) have to match each other.

- (b) There are many different ways of linking the actual parameter to its formal parameter. Thus, in computer science there are several **parameter passing modes**. In this course, you get to know two of them, i. e. the two passing modes supported by the language C++.

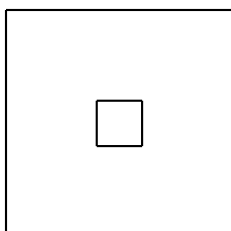
- **Passing by value (‘CALL BY VALUE’):** the actual parameter’s value is copied into the formal parameter’s own memory location; after this, there is no further connection between the actual and the formal parameter.
- **Passing by reference (‘CALL BY REFERENCE’):** the formal parameter is made into a synonym of the actual parameter, into a so-called reference (Referenz). All that is performed with the formal parameter, is really done with the actual parameter.

The consequences of these two parameter passing modes are different according to the access within the function:

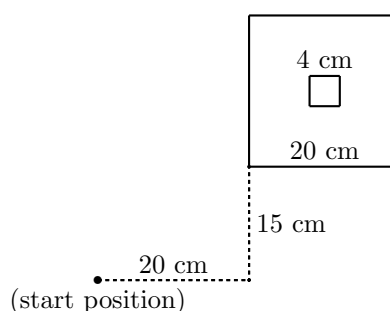
- If the function has only a **read access** on the formal parameter, there is no difference in the result of these two passing modes. Merely, there are differences in behavior with respect to memory and time.
- If, however, the formal parameter is altered within the unit (**write access**), e. g. by assignment, in the case of value passing the actual parameter does not notice anything. But this is different in the case of reference passing: in reality, the actual parameter is altered because the formal parameter is a synonym for it.

- (6.24) Ex A robot is to perform the following task:

Draw the figure below
(two nested squares):



The same with
measurements:



The robot must know the following elementary algorithms (2.54):

new	goes to the defined start position
drop	drops the pen
lift	lifts the pen
linear	motion of one increment of the linear step motor in the actual direction
rotate	motion of one increment of the rotary step motor in the left direction

```

ALGORITHM TwoSquares
PROCEDURE motion (length TYPE Integer)
  // moves pen by length cm in the actual direction
  VARIABLE n TYPE Integer
  n ← length/(one step of the linear step motor in cm)
  REPEAT n times
    linear
  END PROCEDURE

PROCEDURE rotation (angle TYPE Integer)
  // changes translation direction by angle degrees (into math. positive direction)
  VARIABLE n TYPE Integer
  n ← angle/(one step of the rotary step motor in degrees)
  REPEAT n times
    rotate
  END PROCEDURE

PROCEDURE drawSquare (side TYPE Integer)
  // draws square of side length side cm
  drop
  REPEAT 4 times
    motion(side)
    rotation(90)
  lift
  END PROCEDURE

PROCEDURE diagonalMotion (deltaX TYPE Integer, deltaY TYPE Integer)
  // moves pen by deltaX cm in x direction,
  //      by deltaY cm in y direction
  motion(deltaX)
  rotation(90)
  motion(deltaY)
  rotation(270)
  END PROCEDURE

BEGIN
  new
  diagonalMotion(20,15)
  drawSquare(20)
  diagonalMotion(8,8)
  drawSquare(4)
  new
  END

```

Since there is only read access on the formal parameters, the parameter passing mode (value or reference) is insignificant.

The first two procedures are the least complex ones, the two others use the simpler units. The (main) program (between BEGIN and END) uses almost exclusively the more complex procedures. Such a nesting of units is done very frequently.

(6.25) For realizing function construct units in C++ see (ch. 7).

6.3 Storage classes, modular programming

(6.30) Ov In order to solve some tasks, it may be sensible to distinguish between two different storage classes (6.31), i. e. different kinds of life time and memory administration. Memory space existing during the complete program run time belongs to the static storage class. Data of the automatic storage class use memory space only during a part of the program's run time; the memory's administration (reserving and releasing) is automatically carried out by the run time system. In (6.32) you see an example of reasonable use of static and automatic storage space.

After this, the modular programming (6.33) is briefly introduced. In this programming style, several functions are combined with the data to be handled by them. In C++/C such modules are realized by separate program files (translation units). An example in pseudo code makes this module construction clear.

(6.31)

- (a) With the procedural programming however, for example, it is difficult to build a stack memory (Kellerspeicher, Stapelspeicher, Stack). If you want to avoid the (unwanted or wanted) access to the data from outside, you can do this only by constructing a 'function with a memory'. You must permit static storage, which is preserved during the complete program run time, independently of a possible actual access: **static storage class**.

⚠ *Thus, the concept of the function from the last chapter is altered or expanded: a function call depends not only on the parameter values, but also possibly on former function calls. This can be a risk when programming unclearly.*

- (b) Memory locations which are reserved only while executing a function, belong to the so-called automatic memory administration: **automatic storage class**; this is a kind of dynamic storage administration. The programmer does not need to control reservation and release of memory space, this is done automatically by the (run time) system. The variables of all former programs belong to this storage class.

(6.32) In the pseudo code, the word `STATIC` causes memory reservation for the complete program run time (static storage class). Restricting access to statements only within the function—independently of being static—is done by putting them within the function.

Ex Stack constructed with static variables (keyword `ARRAY` see (2.53)):

```
// Definition of the function stack
FUNCTION stack(pushYesNo TYPE Boolean,
               value TYPE Integer) TYPE Integer
  STATIC VARIABLE memory TYPE Integer-ARRAY[100]
  STATIC VARIABLE count TYPE Integer (start value 0)
  IF pushYesNo THEN
    // push here without error handling (overflow)
    memory[count] ← value
    count ← count+1
  ELSE
    // fetch (pop) here without error handling
    count ← count-1
    value ← memory[count]
  RETURN VALUE value
END FUNCTION
```

```

// Using the function stack
BEGIN
  VARIABLE number TYPE Integer
  Read number
  stack(true,number)
  stack(true,199)
  Write stack(false,0) // 199
  Write stack(false,0) // number read
END

```

- (6.33) A disadvantage of the function from (6.32) is that the static array is not available to two separate functions, e. g. `push(value)` and `pop()`; the first one is used for increasing the stack, the second for reducing it. Placing the memory (here as array memory) outside the function would indeed solve the problem; however, an unwanted or wanted direct access to this memory would be possible without using the access functions (which should also include error handling).

The problem can be solved by creating the possibility to encapsulate several functions and their data in the way that only parts of them are public, others are hidden. This is achieved by the **modular programming**, namely by distributing functions and data to several modules which have an additional possibility to restrict or to grant access to each of their parts. This changed view of programming is made by moving the attention from designing functions to organizing data, this kind of decomposition is more data oriented (6.11).

Such a **module** consists of a set of related functions and the data which are to be manipulated by them. Now, the functions can mostly not be independently used or cannot be easily freed from the module (e. g. for other programs), but only as a complete module.

In practice, such modules are normally realized by translation units (Übersetzungseinheiten).

- (6.34) In the pseudo code, a module is represented by its box. Additionally, the access specifiers `HIDDEN` and `PUBLIC` are introduced. Within a box, you can access each member independently of their access specifiers. From outside, you can (directly) access only the public parts. All public members together are called the module's **interface**.

Ex Stack from (6.32), formulated as a module:

```

HIDDEN STATIC VARIABLE memory TYPE Integer-ARRAY[100]
HIDDEN STATIC VARIABLE count TYPE Integer (start value 0)
PUBLIC PROCEDURE push(value TYPE Integer)
  // here without error handling (overflow)
  memory[count] ← value
  count ← count+1
  RETURN
END PROCEDURE
PUBLIC FUNCTION pop() TYPE Integer
  // here without error handling
  count ← count-1
  RETURN VALUE memory[count]
END FUNCTION

```

```

// Using the stack
BEGIN
  VARIABLE number TYPE Integer
  Read number
  push(number)
  push(199)
  Write pop() // 199
  Write pop() // number read
END

```

- (6.35) How to use the different storage classes and how to realize modules (translation units) in C++ see (ch. 8).

6.4 Object oriented programming

- (6.40) Ov The object oriented programming, just as the modular programming, allows the encapsulation of functions and data. In contrast to modular programming where each module contains exactly one set of data, here you have to distinguish between description of the data's and functions' structure, the so-called class definition, and the actual creation of the memory space at run time, the generation of the so-called objects. Here you may generate, and use, and destroy as many objects of the same type of construction (the same class) as you want to.

The encapsulation is well made when data ('attributes') are never directly accessible from outside, but always only via functions belonging to them ('methods') which, in addition, control this access, e. g. to guarantee the internal data's consistency.

Additionally, object orientation is explained in more detail by an example (6.43).

- (6.41) Modular programming (see last subchapter) indeed enables you to define an access control from outside. But a very important disadvantage is not yet removed: it is not possible to have several independent stacks. A makeshift could be introducing several hidden arrays; but the maximum number of simultaneously used stacks must be fixed before.

(6.42)

- (a) **Object oriented programming** solves the problem: a new type **class** (Klasse) is introduced. This type encloses data as well as their access functions, to be precise, each data or function member with an explicit access control **HIDDEN** or **PUBLIC**. Within the class, you have access to all members, also to hidden ones, from outside only to public ones. A variable of this type is called an **object** (Objekt). Each object has its own set of data. In contrast, the access functions normally exist only once for each class.

Independently of a special language, the data of a class are mostly called **attributes** (Attribute), the functions mostly **methods** (Methoden). In C++ they are also called (**data**) **members** and **member functions** respectively.

- (b) With respect to a good distinction between hidden and accessible (6.13), one should mark **all attributes** as **HIDDEN** and allow access to them only via public access functions.
- (c) The above described abstractions are actually not enough to earn the name 'object oriented'. Additionally, the expansion by **inheritance** indicated in (6.45a) is very important.

↑↑1 *Since the beginning, programmers have the idea of separating data and the functions belonging to them. A human not infected by this kind of programming technique would not get the idea of doing so. In the real world, data and their functions (i. e. the state's description and the functionality) almost always belong to each other, e. g. a car has a maximum speed, a maximum gas volume, an average gas consumption (these are data); additionally it moves, it stops, it consumes fuel (these are functions). Both data and functions belong to a real car.*

↑↑2 *The transition to object orientation (or to **object technology**) should not be constrained only to **object oriented programming (OOP)** (objektorientierte Programmierung). If you reasonably want to take advantage of these ideas, this is only possible if you begin the new viewpoint already long before the programming (or implementation). Firstly, the part of the real world which is to be controlled by the new software is to be described in the sense of this new viewpoint (**OOA, object oriented analysis** (objektorientierte Analyse)). Then the resulting model is to be changed or expanded to meet software requirements (**OOD, object oriented design** (objektorientierter Entwurf, objektorientiertes Design)). In order to express this transition, some authors use the terms: changing the kind of world view, changing the conception of the world. They call it a 'paradigm shift' in the designer's mind, cp. (6.11).*

↑↑3 *The idea of modelling in the object technology is to describe the real world as a network of interacting objects. Each object (as a specimen built according to a class's construct pattern) has a certain independence and **responsibility for itself**. A **message** (Nachricht) sent to it, is regarded by the object as a (polite) request to react according to its functionality. To do so, it executes one of its methods. But the kind of reaction underlies its responsibility because the method belongs to it, cp. (6.11).*

↑↑4 The kind of world view in object technology is more **integrated**; it lets objects communicate with each other as units. Many programmers who have the traditional programming view point have great difficulties to perform the paradigm shift.

↑↑5 It is a pity, but in this course you will learn very little about this world view (paradigm); perhaps this can be deepened in later courses.

(6.43) Ex Stack from (6.34), designed object oriented:

```

CLASS Stack
  HIDDEN VARIABLE memory TYPE Integer-ARRAY[100]
  HIDDEN VARIABLE count TYPE Integer (start value 0)
  PUBLIC PROCEDURE push(value TYPE Integer)
    // here without error handling (overflow)
    memory[count] ← value
    count ← count+1
    RETURN
  END PROCEDURE
  PUBLIC FUNCTION pop() TYPE Integer
    // here without error handling
    count ← count-1
    RETURN VALUE memory[count]
  END FUNCTION
END CLASS

```

```

// Using this stack:
BEGIN
  VARIABLE stack1, stack2 TYPE Stack
  VARIABLE number TYPE Integer
  Read number
  stack1.push(number)
  stack2.push(199)
  Write stack1.pop() // number read
  Read number
  stack2.push(number)
  Write stack2.pop() // number read
  Write stack2.pop() // 199
END

```

You see the new keyword CLASS (and END CLASS); by this, a class is defined. Invoking an object's method is done by the syntax

ObjectName.MethodName (...) .

(6.44) How to realize class construct units in C++, see (ch. 9).

(6.45) ↑↑

(a) Introducing the class type, i. e. the encapsulation of data and functions, is not sufficient to describe the real world simply and clearly enough. Therefore, some authors call the aforementioned ideas 'object based programming', the data types belonging to it are sometimes called 'abstract data types'. The object technology gets powerful only by introducing the possibility to describe the similarity of classes: the **inheritance** (Vererbung). A super class (Oberklasse), in C++ called base class (Basisklasse), passes its properties (data and functions) to a sub class (Unterklasse), in C++ called derived class (abgeleitete Klasse). This sub class is allowed to expand or overwrite (replace) these properties.

(b) Sometimes it is helpful to have classes for which the member type is not yet settled; settling it is done only if the class is used: **template** (Schablone, Template). Templates are important especially for container classes (or shortly **containers**), for classes which administrate a set of elements of a type.

Inheritance will be explained in (ch. 11.3), but templates cannot be described later.

7 Procedural programming

7.0 Overview

Procedural programming is language-independently described in more detail in (ch. 6.2). It mainly uses only procedures and functions as construct units.

A function or procedure—both called functions in C++—is a collection of program code with a name, so that this code can be executed several times only by calling the name. Mostly in addition, there are also actually settable parameters. Subch. 1 describes how to define (build) such a function and how to call (use) it.

Since in C++ there is the rule that you have to introduce a name to the compiler before using it, in some cases the correct order (a function’s definition before its first call) may be tricky. Therefore, very often the possibility of introducing functions without showing the function code is used. This is called function declaration (subch. 2).

The reference allows you to generate synonymous names for already existing variables (subch. 3). This is not used very frequently, in general, however more often in the special case of function parameters (subch. 4). Here you find a detailed description of the two C++ passing modes for function parameters (value and reference) that have to be distinguished strictly, and how to apply these two modes sensibly.

The scope of names may be local (block related) or global (file related). Subch. 5 describes this fact and renders very important programming hints about which names should be global and which local.

After some supplements about functions in subch. 6, subch. 7 introduces the field of function’s recursion. In some cases, such a recursion can make programming easier. It may be that at first you will have problems to understand this, therefore you should follow the explanations in the lecture very carefully.

The chapter concludes with some hints on how to make functions so that they are of good quality.

7.1 Function definition and function call, scope within blocks

(7.10) Ovw At first, you will learn to distinguish:

- **Function call:** the function’s use, here you need only the answer to the question ‘What?’ (What does the function achieve?), see (7.12), cp. (6.12).
- **Function definition:** the function’s construction, here the question ‘How?’ is answered (How does the function realize the demanded feature?), see (7.11).

Point (7.13) renders an example program with function definitions and function calls.

To invoke a function means a great administrative effort for the processor during run time. The single steps required to do this are described in (7.14). It is useful to know this, but for programming itself this is less important, mostly.

The actions to be done when a function is executed are written within a *CompoundStatement*, the function block. Therefore (7.15) renders the rules for the scope of names in blocks, also e. g. in nested blocks.

(7.11)

- (a) $^{10/33} \textit{FunctionDefinition}$ (special *Declaration*, simplified) \doteq
 $^{12,14} \textit{ReturnType}$ $^{30,32(\textit{Alt.4})} \textit{FunctionIdentifier}$ ($\textit{ParameterDeclaration-LIST}_{opt}$)
 $^{54} \textit{CompoundStatement}$
- $\textit{ParameterDeclaration-LIST}$ (simplified) \doteq
 $\textit{Type FormalParameter}$ [, $\textit{Type FormalParameter}$] $_{0..n}$
- $\textit{FormalParameter}$ \doteq *Identifier*

The *ReturnType* sets the type of the function's return value; according to (6.21b) you have to distinguish between two kinds of functions:

- Function with return value: 'normal' return type. At the calling location, the function name contains the type and its value (the latter: main effect of the function expression).
- Function without return value: type `void`.

Ex see (7.13)

Note The *CompoundStatement* is the function block.

↑↑ c ⚠ In the language C, a function without parameter should be defined with the keyword `void` instead of an empty parameter list, details see (7.23 Note4).

- (b) The return jump out of the function is made by an additional statement, a special ⁵³ *Jump-Statement*; it can appear at several locations in the function:

```
return Expressionopt ;
```

If the program control flow encounters a `return` statement, the function is left directly:

- In a function with return value, *Expression* must not be omitted, for it sets the actual return value.

Note To improve clarity, the *Expression* is often enclosed by (); this is syntactically allowed, but not necessary.

- In a `void` function, *Expression* must be omitted. In this case, a `return` statement can be left out at the end of the function because it is implicitly included by the compiler.

- (7.12) The call (i. e. the use) of a function is made by an expression:

$$\text{FunctionCallExpression} \doteq \text{FunctionIdentifier} (\text{Parameter-LIST}_{opt})$$

$$\text{Parameter-LIST} \doteq \text{ActualParameter} [, \text{ActualParameter}]_{0..n}$$

$$\text{ActualParameter} \doteq \text{Expression}$$

When a function is called, the control flow jumps to the function (to be precise, to the beginning of the function block); it returns to the location where the function was called when it encounters a `return` or the end of the function block.

The *FunctionCallExpression* can have two effects, cp (3.31):

- Side effect (always present): jump into the function, executing the code.
- main effect (value of the expression):
 - Main effect exists if you have a function with return value, namely the value of the expression of the `return` statement where the function was left.
 - Main effect is missing if you have a function without return value, i. e. in a `void` function. Therefore, a function call without a return value is allowed only where the expression's value is ignored, e. g. in an *ExpressionStatement* (3.32a).

Correspondence principle, concerning the *Parameter-LIST*: number, type, and order must match the formal parameters, if necessary after implicit type conversion.

Ex see (7.13)

- (7.13) Always, the control flow of a C++/C program begins with execution of the main function.

Ex Example program for function definition and function call

```
#include <iostream>           // example program E07-13.CPP
using namespace std;

int valuePlus1(int value)     // -- shorter, but equivalent: --
{
    int result;              //    int valuePlus1(int value)
    result=value+1;          //    {
    return result;           //        return value+1;
}                             //    }

int sum(int par1, int par2)   //    int sum(int par1, int par2)
{                             //    { return par1+par2; }
    int result;
```

```

    result=par1+par2;
    return result;
}

int numFixed()                //      int numFixed()
{                             //      { return 4711; }
    int value;
    value=4711;
    return value;
}

void intPrint(int value)      // funct. call: expr. without main eff.
{ cout << value << endl; }

int main()                   // also a function definition!
{                             // HERE begin of the program's control flow
    int i, j, k, value;
    i=5; j=21;

    cout << valuePlus1(i) << ' ' << valuePlus1(j+3) << endl; // 6 25
    k=sum(i,j);
    intPrint(k);              // 26
    intPrint(sum(5,i)+sum(j+1,-16)); // 16 (10+6)
    value=numFixed()+5;
    intPrint(value);         // 4716
    intPrint(2+value+numFixed()); // 9429

    return 0;
}

```

(7.14) ↑↑ Order of events when calling a function

Note *The order of events can be different depending on the programming language. It is explained here as it occurs in C++/C. The parameter passing mode is assumed to be value, see (6.23b). The order when passing by reference is similar.*

- (a) If actual parameters exist:
- (1) All actual parameters are evaluated, additionally, if necessary, type conversion to the type of the corresponding formal parameter.
 - (2) Memory space for the formal parameters is reserved, the calculated values are copied into these locations: this is the formal parameters' initialization by the actual parameters.
- Note *Actual and formal parameters are to be regarded strictly separated, even if you have (random or intended) name equality. It is certain that all possibly existing side effects are executed before the following step, cp. (3.33).*
- (b) The function call's address is stored for the return jump (more exactly: the address of the following action).
 - (c) Jump to the beginning of the function block.
 - (d) If there are local variables: memory space for them is reserved.
 - (e) The function code is executed until control flow encounters a **return** statement or the end of the function block.
 - (f) If a **return** expression exists: the expression is evaluated, then type conversion to the function's return type (if necessary), after this, this value is stored temporarily: before this, all of the expression's side effects are executed (if existing).
 - (g) If there are local variables: their memory space is released, cp. (d).
 - (h) Return jump to the saved call location, cp. (b).
 - (i) If there were parameters: the formal parameters' memory space is released, cp. (a2).
 - (j) If the function had a return value (f): the value of the function's name at the call position is the saved **return** value.

(7.15) Rules for **scope** in blocks, resolution of name conflicts


(supplement to (4.12), summary see (ch. 7.5); exceptions to some of these rules see the remarks)


- (a) Names declared in a block (⁵⁴*CompoundStatement*, e. g. function block, but also in a subordinate block) are called **local names**.


The **scope** (Gültigkeitsbereich) of a local name ranges from its declaration point (see [DeclPoint] in ¹¹) until the end of this block.

Within the block level that it is declared in, the name must be unique, it must not be declared differently several times (more exactly: see (Note1)).

- (b) A name is **hidden** in subordinate blocks by declaring it there once more; when leaving this block, the name in its old meaning (if variable: with its old value) is available again.


- (c)  Variables defined within a block do not have any defined value when entering this block! Thus, giving a value—this can be done by initializing or e. g. by assigning a value—before the first reading is absolutely necessary!!

 Exceptions to this rule are objects if they have well built constructors (9.21), furthermore static variables (8.12).

 **Ex** Foolish function because `risk` receives a random value:


```
int nonsense(int value)
{ int risk;
  return value+risk; // nonsense! risky!! no!!
}
```


- (d) Reserving the memory space for local variables of the aforementioned kind is done automatically when entering the block; when leaving this block, the space is automatically released—therefore the name ‘automatic’ variables (cp. (7.14 d,g) in function blocks) or automatic storage class (6.31b). The reservation is preserved during execution of subordinate blocks, too, even if its name is hidden.


 For reservation of the memory space for local *static* variables see (8.11, 8.12), additionally also (6.31a).


- (e) The **formal parameters** are regarded as defined in the outer function block with respect to the above rules (scope, uniqueness, hiding ability)


Important difference to the automatic variables: the formal parameters are initialized by the actual parameters, after entering the functions they have initial values.

 **Note1** Multiple, non-contradictory declarations are allowed (but no multiple definitions), as for functions, see (7.23 Note3).

 **Note2** A definition is allowed to occur only once at the same block level.

 **↑↑** Certain exceptions are allowed if you use enumerations and classes/structs, see (C++/5.1).

 **Note3** There is no name conflict with possible identical names within other functions.

- (f)  **Ex** The following example program is not a good programming example because it is very confusing: unnecessary block nesting and multiple definition of the same name. But nevertheless, it is important to comprehend it; therefore it serves only the purpose of learning, of exercising. The denotations **xxx** in the comments indicate the different memory spaces with the same name.

```
#include <iostream>           // example program E07-15.CPP
using namespace std;

void print(int value)
{ cout << value << endl; }

int main()
{
  int abc;                   // *abc1*
  abc=5;                     // *abc1*
  print(abc);                // 5 *abc1*
}
```

```

    int abc=9;           // *abc2*
    {
        double abc;     // *abc3*
        abc=3.7;        // *abc3*
        // implicit type conversion (cut),
        // therefore possibly compiler warning:
        print(abc);     // 3 *abc3*
    }
    print(abc);        // 9 *abc2*
}
print(abc);           // 5 *abc1*
{
    int def;
    def=10;
    print(def);        // 10
    print(abc);        // 5 *abc1*
}
// The following line could not be compiled:
// print(def); // error! variable "def" no longer available!
print(abc);           // 5 *abc1*

return 0;
}

```

7.2 Function declaration

(7.20) Ov A function’s declaration informs the compiler of a function’s name. In C++/C—in contrast to e. g. Pascal—it is usual to declare functions. In this case, the definition may appear elsewhere.

(7.21)

- (a) A **declaration** introduces a name for the compiler; the compiler gets to know the necessary properties for using this name.
- (b) A **definition** is a special declaration; here the compiler gets to know everything about the name, not only how to use this name.

(7.22)

- (a) A **function definition** introduces the function name for the compiler; additionally memory space is reserved for the function code.
- (b) A function definition is made normally in another program file (see (ch. 8.2), e. g. also in a library), or in the source code after the first call. But the compiler has to know the meaning of the name, at the latest at the first function call.

That is the purpose of a **function declaration**; it introduces the name and informs the compiler about the function’s parameters and return type.

Function declarations are used very often in C++/C, even if they can be avoided because the definition could occur before the first use.

(7.23) $^{10/11} \textit{FunctionDeclaration} \doteq \substack{12/14 \textit{ReturnType} \\ 30/32, \textit{Alt.4} \textit{FunctionIdentifier} \textit{ (ParameterDeclaration-LIST}_{opt} \textit{) } ;}$

A function declaration is syntactically very similar to a function definition, see (7.11a), instead of the function block (*CompoundStatement*) there is a semicolon.

With such a function declaration, the compiler gets the information about:

- the function’s name,
- the function’s signature (number, type, order of the parameters),
- return type.

Only the information about the function’s entry address is missing, see (7.14c). This entrance address must be set later on:

- by the compiler if the definition occurs later in the same program file, or
- by the linker if the function is defined elsewhere.

Ex Declaration of all functions from example program (7.13):

```
int valuePlus1(int value);
int sum(int par1, int par2);
int numFixed();
void intPrint(int value);
```

Note1 In the header files, there are many function declarations, e. g.

```
<iostream>: get, getline;
<cstring>: strcpy, strcat, strlen.
```

Note2 In a declaration (normally not in a definition), it is permissible to leave the formal parameters' names, not the types.
 Recommendation: do not leave out the names because they (hopefully!) describe the meaning of the parameter ('self documenting name').

Example:

```
double powerFunction(double base, double exp); // recommended
double powerFunction(double, double); // allowed - but meaning of the par.?
```

Note3 A function declaration may occur multiple times as long as the declarations do not contradict each other (the same name, return value, signature; the parameters' names can be different; a different signature indicates a different function, cp. (7.61)). It is allowable to have a declaration of a non-used function. On the other hand, a function definition is allowed exactly once only; if the function is not used, this definition can be left out.

↑↑ With virtual member functions (of a class), there must be a definition even if the member function is not used, see (11.42↑↑1).

Note4 **C** ⚠ If there is no parameter: in any case, the definition and the declaration should have the following form

```
Return Type FunctionName(void)
```

instead of

```
Return Type FunctionName() // empty parameter list
```

The use of `void` is allowed in **C++**, too, but it is not necessary; in **C** it is very important because the empty parantheses' pair has a different meaning because of historical reasons (function with a non-specified number of parameters).

7.3 Reference type

(7.30) **Ov** A reference creates a synonym of an already existing variable, i. e. an additional name referencing the same memory space as the original name. In this subchapter you see how to generate and to use a reference in C++. In practice, this is done nearly only in combination with functions, namely as function parameters; this is described in the following subchapter.

(7.31) New type: **reference type**.

(a) With a reference, a synonym (an additional name, an alias) with an already existing variable is generated. Therefore, it is absolutely necessary to distinguish between these two steps:

- **Initialization** of the reference, i. e. setting the reference; with this, the new name is linked to an already existing variable.
- **Using** the reference, to be precise, as a synonym with this other variable.

DefinitionReferenceWithInitialization _[NC] \doteq *Type &Identifierr* = *OriginalVariable* ;

Type of reference: as the type without the character `&`.

Ex Program fragment with definition and use of reference:

```
int i;
int &j=i; // setting the reference, from now on i and j synonymous
// both definitions shorter: int i, &j=i;
i=5;
cout << j; // 5
j=34;
cout << i; // 34
```

Note1 About the declaration operator & see ³¹.

Note2 A reference must always be initialized in order to announce to the compiler which variable it shall be synonymous with.

↑↑ Indeed, a declaration without initialization is possible, too, if the initialization occurs elsewhere—but this possibility is not discussed here, it is rarely used.

- (b) A reference can be constant (`const` before *Type* in the definition (7.31)); this means that with this reference, only a read access, no altering access is allowed. Application: as a function parameter (7.44b1).

7.4 Parameter passing modes value and reference

- (7.40) Ov When using functions, it is essential to know how an actual parameter is linked to the corresponding formal parameter. Therefore you have to know the two **parameter passing modes value and reference** in C++, and how to apply them. This subchapter explains in detail when and how to use these two modes. If you use array names as parameters, you have to observe some important specialities (7.46).

- (7.41) Depending on the programming language, you can differentiate several parameter passing modes, i. e. ways to link actual and formal parameters of a function.

In C++, there are two passing modes, in C only one mode; general description of these modes see (6.23b).

- (a) c/c++ Passing by value (‘CALL BY VALUE’): the actual parameter’s value is copied onto the formal parameter, see (7.42).
- (b) c++ Passing by reference (‘CALL BY REFERENCE’): a reference to the actual parameter is passed, see (7.43).

**In both parameter passing modes:
the formal parameter is initialized by the corresponding actual parameter,
it is no assignment.**

Except for this—important—difference, formal parameters are like local variables of the function block, see (7.15e).

- (7.42) Passing mode **value** (‘CALL BY VALUE’) C/C++

The actual parameter’s value is copied onto the formal parameter’s own memory space (that is, in this case, the meaning of initialization). Since the value is copied, the actual parameter does not need to have a data memory address, it can be an arbitrary (arithmetic) expression. After copying, there is no link between formal and actual parameter any more. If you alter the formal parameter’s value (own memory location), the actual parameter is not changed.

Ex see (7.45).

- (7.43) Passing mode **reference** (‘CALL BY REFERENCE’) C++

The formal parameter becomes a synonym with the actual parameter within the function (that is, in this case, the meaning of initialization). Each alteration of the formal parameter’s value implicitly acts on the actual parameter. As generally described for the reference type in (7.31), here you must also distinguish the two steps initialization and use:

- Initializing (setting) the reference is performed by each function call, the formal parameter is initialized with the (current) actual parameter.
When applying this passing mode, the actual parameter must be a variable, i. e. it must have a data memory address, it must not be a (general) expression. Exception: constant reference (7.44b).
- Using the reference is done by using the formal parameter within the function block.

Definition $\text{FormalParameterReferenceType}_{\text{[NC]}} \doteq \text{Type \&Identifier}$

The initialization is done by the function call, in contrast to this cp. (7.31).

Ex see (7.45).

Note With this passing mode, it is possible to receive more than one value from a function.

↑↑ C/C++ There is another way of receiving altered values from a function: (explicit) use of pointers—despite of CALL BY VALUE, cp. (5.65a).

Using an array as a function parameter, you must know some specialities, see (7.46).

(7.44)

- (a) Unfortunately, you cannot recognize from the syntax of a function call whether there is a value or a reference passing mode. This is obvious only in the function’s declaration or definition.

Therefore there is no clarity or comprehensibility at the place of the function call if you do not know whether the actual parameter can be changed by the function call or not, unless the actual parameter is not or not only a variable (then no changeable reference is possible). Thus there are authors who recommend avoiding a changeable reference. But this no deep problem for persons who learnt C++ without former deep C knowledge, because they are used to encountering references.

- (b) Which kind of parameter passing mode is to be used depends at first on the inquiry, if the parameter will not (or should) not be altered by the function call (1), or if you want to receive a changed value (2).

- (1) If a **parameter is not to be altered by the function** (nor by an inadvertant program bug), you should follow these hints:

- The **normal passing mode** should be **CALL BY VALUE**, especially with built-in types.
- C++ Sometimes however, especially when the variable demands a large memory space, you want to save the extra memory and the copy time of a CALL BY VALUE at run time. Yet nevertheless, you want to be sure that the actual parameter cannot be changed, nor by an inadvertant program bug in the function. The solution is the **constant reference** (7.31b). This is applied very often with objects (ch. 9).

Syntax:

DeclarationConstantReference _[NC] \doteq ^{14/20} **const** *Type & Variable*

In a constant reference, the actual parameter is allowed to be an expression, possibly of another type (intermediate storage in a temporary variable).

- (2) If a **parameter is to be altered by the function**, you have to do:
- use the (non-constant) **reference passing mode**,
 - or another possibility (pointer), further discussion see (12.42, 7.43).

Ex see (7.45).

- (7.45) Ex Example program for parameter passing mode value and reference (non-constant and constant); in the lecture, the memory map will be explained.

```
#include <iostream>           // example program E07-45.CPP
using namespace std;

// Function declarations; definitions see below
void swapNotMuch(int par1, int par2);
void swap(int &par1, int &par2);
int newValue(int par1, const int &par2, int &par3);

int main()
{
    int i, j, k;

    // Swap no and yes:
    i=5; j=7;
    swapNotMuch(i,j);
    cout << i << ' ' << j << endl;           // 5 7
```

```

    swap(i,j);
    cout << i << ' ' << j << endl;                // 7 5

    // constant and non-constant reference:
    i=12; j=32; k=-5;
    cout << newValue(i,j,k) << endl;                // 46
    cout << i << ' ' << j << ' ' << k << endl;        // 12 32 -3

    cout << newValue(i+6,j*2,k) << endl;            // 86
    // newValue call: general expressions permissible as parameters?
    // 1st par. (value): anyway
    // 2nd par. only because reference is constant
    // 3rd par. NO, only variable because mutable reference
    cout << i << ' ' << j << ' ' << k << endl;        // 12 32 -1

    return 0;
}

void swapNotMuch(int par1, int par2)
{
    int note;
    note=par1;
    par1=par2;
    par2=note;
}

void swap(int &par1, int &par2)
{ // program code identical to "swapNotMuch()"
    int note;
    note=par1;
    par1=par2;
    par2=note;
}

int newValue(int par1, const int &par2, int &par3)
{
    par1+=5;    // permissible, however actual parameter unchanged

    //par2+=8;  // would NOT be permissible because constant;
                // namely: actual parameter would be changed

    par3+=2;    // permissible because non-constant reference

    return par1+par2+par3;
}

```

(7.46) Specialities using arrays as function parameters C/C++

Using an array name as a parameter, the **passing mode seems to be reference** (more exactly see ↑↑), i. e. the effect is like passing a reference. Therefore it is possible to alter arrays as function parameters, cp. `strcpy`, `strcat` (5.53ab). A real CALL BY VALUE of an array (in the sense of a local copy within the function) is not possible in C++/C. Thus, you should use a `const` if the function should not alter the array.

The type denotation of the formal parameter may be the actual array type (without any symbol `&`), the number of elements (but not the pair of square brackets) may be left out. In (12.44a) this will be discussed in more detail; there you will also find the more common kind of type denotation.

Ex Function declaration, the parameter is an `int` array with 20 elements (in the case of function definition, instead of the semicolon there has to be the function block):

```
void doSometh(int arr[20]); // function declaration
```

The following declaration would have the same meaning (the number 20 is left out):

```
void doSometh(int arr[]);
```

A function call (e. g. within `main()`) with prior definition of a suitable array:

```
int manyInteger[20];
doSometh(manyInteger);
```

Certainly, using a symbolic constant (5.44) would be much better than the above program text:

```
const int numIntArr=20;
void doSometh(int arr[numIntArr]);
```

or without the number within the square bracket pair (as above):

```
void doSometh(int arr[]);
```

Call, e. g. within `main()`:

```
int manyInteger[numIntArr];
doSometh(manyInteger);
```

If the array should not be altered in the function, use a `const` prior to the array type:

```
void doSomethWithoutChange(const int arr[20]); // function declaration
```

or (the number 20 is left out):

```
void doSomethWithoutChange(const int arr[]);
```

There is no change in the function call:

```
int manyInteger[20];
doSomethWithoutChange(manyInteger);
```

↑↑ Reasons in more detail, cp. (10.26) and (12.44):

Since an array name (after its declaration) is (nearly) always automatically converted to a pointer to its element number 0, an array name's use looks like a *CALL BY REFERENCE*. But actually, it is a *CALL BY VALUE* of a pointer, cp. (5.65b).

So, in reality, a pointer is passed to the function. This is also the reason why within a function, the expression

```
sizeof(ParameterArrayName)
```

is interpreted very differently from the (possibly) first idea: it is the required memory space of the pointer (e. g. 4 bytes), and not the required memory space of the original array, see (12.44d).

7.5 Global and local names

(7.50) Ow At first, the terms **local name** (valid only within a block) and **global name** (valid in nearly the complete program file) are explained (7.51). A simple nesting of blocks is made possible by hiding (7.52) because, in this way, name conflicts can be avoided to a large extent. To have a good programming style, it is essential to know that variables should (nearly) never be global, in contrast to this, global constants and types are often very sensible (7.53).

(7.51) Global and local names

- (a) Names (e. g. of variables or types) are called **local** (formerly also: internal) if they are declared within a block, cp. (4.12) and (7.15a). Their **scope** (Gültigkeitsbereich) ranges from their declaration point (directly behind the names, more exactly: behind the declarator, see ¹¹) until the end of the block they are declared in. The formal parameters of a function are regarded here as declared in the outermost function block, i. e. they are local names (7.15e).

⚠ Pay attention—already mentioned in (4.13): in general, local variables (if automatic, details see (8.13)) are **not automatically initialized**, so you should initialize them explicitly or assign them a value shortly after definition.

- (b) Names (e. g. of variables or types) are called **global** (formerly also: external) if they are declared outside of blocks. Their **scope** extends from their declaration point until the end of the translation unit.
- (c) **Class members** have the **scope** class, see (9.12a).

(7.52) Names—in the level of their declaration—must be unique. In contrast to this, a (local or global) name is **hidden** by the declaration of the same name in a subordinate block; after leaving this block, the name with its old meaning is accessible, cp. (7.15b).

C++ Global names—independently of being hidden—are always accessible with the aid of the scope resolution operator `::` (Op1a); but a local name’s hiding cannot be broken.

Note *Summary see also (C++/5.1);* **↑↑** *in the note there, you can also read about exceptions to this uniqueness rule.*

(7.53) General guidelines for creating and using global names:

- **Global variables** are hard to understand because they may be altered anywhere. Therefore, they are **not permitted** (in this course). In general, they should be avoided wherever possible.
- In contrast to them, **global constants** are used often because their values cannot be altered.
By the way, remember: use symbolic constants instead of constant numbers wherever possible, see (4.72), and (5.44).
- **Type names** often are global so that they can be used in the complete program file. The meaning cannot be altered, so having global types is no problem.
- **Function names** are always global because C++/C does not allow local functions (in contrast to e. g. Pascal).

↗ When several source text files are used (ch. 8.2), function names should be hidden whenever possible. Detailed general considerations see (8.21), additionally application for header files with respect to constants, types, functions etc. see (8.24).

7.6 Overload of function names, default arguments

(7.60) **Ovw** In this subchapter, two additional specialities of C++ are presented: You may declare several functions with the same name, the distinction is made by the compiler with the aid of the signature. In addition, it is possible to assign default values to formal parameters which are used automatically when the actual parameters are missing.

(7.61) Identifying a special function:

c The function’s name must be unique (within its scope).

C++ The function’s name including the signature (i. e. number, type, and order of the parameters) is used to select the correct function code; so several functions with the same name are allowed here if they have different signatures (**overloading** function’s names). This property is frequently used, e. g. with constructors (9.21)—but not only in this case.

Ex

```
#include <iostream>           // example program E07-61.CPP
using namespace std;

void output(int value);
void output(double value);
void output(int value1, double value2);

int main()
{
    output(3);
    output(3.);
    output(9,5.3);
    // 2x implicit type conversion, 1st possibly with compiler warning:
    output(5.6,2); // as:  output(5,2.0);
    return 0;
}

void output(int value)
{ cout << "int:      " << value << endl; }

void output(double value)
{ cout << "double:   " << value << endl; }
```

```
void output(int value1, double value2)
{ cout << "int/double: " << value1 << ' ' << value2 << endl; }
```

Here the output of the above program:

```
int:      3
double:   3
int/double: 9 5.3
int/double: 5 2
```

(7.62) Default arguments (Standardargumente) of functions C++

It is permissible to assign a value to one or several formal parameters—to be precise, in reverse order—in the function's declaration (or the definition if it appears before). In a following definition (or declaration), these default values must not be repeated.

If such a value is missed in a function call (parameters missing always from back), the default value is used. Thus, this is no overload of function names (7.61) because the compiler automatically inserts missing values.

Ex1 Function definition with default arguments:

```
void print(int value=4711)
{ cout << value << endl; }

int main()
{
    print(16);      // 16
    int j=15;
    print(j-7);    // 8
    print();       // 4711
    print(4711);   // 4711
    return 0;
}
```

The same with function declaration:

```
void print(int value=4711); // declaration: WITH default argument

int main()
{
    // ... (as above)
}

void print(int value) // definition: WITHOUT default argument!!
{ cout << value << endl; }
```

Ex2

```
#include <iostream> // example program E07-62.CPP
using namespace std;

void toPrint(int value1, double value2=0.7, int value3=-1);

int main()
{
    // 1 parameter:
    toPrint(16); // as toPrint(16,0.7,-1);

    // 2 parameters:
    toPrint(-1,.9); // as toPrint(-1,0.9,-1);
    // also with implicit type conversion (2nd par.):
    toPrint(3,2); // as toPrint(3,2.0,-1);

    // 3 parameters:
    toPrint(123,6.,88);
    // also with implicit type conversion (2nd par.):
    toPrint(123,6,88);

    return 0;
}
```

```

void toPrint(int value1, double value2, int value3)
{
    cout << value1 << ' ' << value2 << ' ' << value3 << endl;
}

```

7.7 Recursive functions

- (7.70) Ow A recursive function refers to itself, i. e. it invokes itself in the function body. At first, this reference to itself seems to be senseless, but it opens short and very clear solutions paths for some problems (7.74). In order for a recursion to be sensible, for each call (i. e. for each nested invocation level) a separate set of data (parameters and/or automatic variables, see (↗)) independent of the other calls must be generated. In order not to run into an infinite recursion, it has to finish some time; this means that the self call within the function is normally controlled by a condition so that after a finite number of nested calls there will be no further self call.

↗ *Automatic variables are local variables, the memory space of which is generated automatically when entering a (function) block and is released automatically when leaving it (6.31b, 8.11b, 8.13). (Until now, no other variables are introduced.) The formal parameters of a function show the same behaviour in this respect.*

If there will be sufficient time in the lecture, the extensive effort for processor administration of the memory will be explained (administration of a stack (7.72a)) with the aid of the simple example (7.71)..

- (7.71) What does the following program section do?

```

#include <iostream>          // example program E07-71.CPP
using namespace std;

const char end='$';

void strange()
{
    char c;
    cin.get(c);
    if (c!=end) strange();
    cout << c;
}

int main()
{
    strange();
    cout << endl;
    return 0;
}

```

- (7.72) Kinds of memory organization:

- (a) In order to execute a recursion as in (7.71) correctly, you need a **LIFO memory** (‘last in—first out’): **stack** (Kellerspeicher, Stapelspeicher, Stack).

Executing such a program recursion is very costly; for each recursion step, you need a function call including automatic (7.70/↗) variables’ reservations and return jump.

Details about constructing a stack memory see the examples in (ch. 6.3/4).

- (b) In contrast to this, a FIFO memory (‘first in—first out’) is a queue (Warteschlange) or a shift register (in electronics).

- (7.73)

- (a) Rule in computer science: Each recursive program structure can be converted into an iterative one, and vice versa.

Since a recursion is very costly at run time, at first you should choose an iterative solution. But if this would mean constructing a stack, you can use the stack implicitly contained in a recursion solution.

- (b) Further examples for program recursion:

Calculating the factorial function,
calculating the Fibonacci numbers.

For these problems, there are simple iterative solutions, too.

Ex Recursion, see also figure in (1.34a): Directory may contain any number of Directory (as specialization of AbstractFile); these themselves may contain also elements of Directory, etc.

- (7.74) A good example for a sensible recursive solution: **towers of Hanoi**.

- (a) Given three places `place1`, `place2`, `place3`. At one of these places, there are `maxHeight` (original: 64) discs of different diameter, sorted in such an order that an arbitrary disc always lies on a larger one or directly on the base.

Task: transfer the disc tower from `place1` to `place2`, according to the following rules:

- you are allowed to move only one disc at a time,
- a disc is only allowed to lie at one of the three places,
- it must always lie on a larger disc or directly on the base,
- `place3` and the other places can be used arbitrarily as an intermediate storage place (clipboard), so long as the above conditions are not violated.

- (b) Approach:

```
const int maxHeight=64;    // Value 64: might take too long??
typedef int Place;        // Place as a type
const Place place1=1, place2=2, place3=3; //for better readability 3 places

void moveDisc(Place fromPlace, Place toPlace)
// Uppermost disc shall be moved from place fromPlace to place toPlace
{
    // must be implemented (robot, screen display or similar)
}

void transpTower(int actHeight, Place fromPlace, Place toPlace,
                 Place helpPlace)
// Tower of the height actHeight (or regarded from above: tower of the
// depth actHeight) shall be transported from place fromPlace to place
// toPlace; place helpPlace can be used as a clipboard
{
    if (actHeight==1)
        moveDisc(fromPlace,toPlace);
    else {
        transpTower(actHeight-1,fromPlace,helpPlace,toPlace);
        moveDisc(fromPlace,toPlace);
        transpTower(actHeight-1,helpPlace,toPlace,fromPlace);
    }
}

int main()
{
    transpTower(maxHeight,place1,place2,place3);
    return 0;
}
```

- (c) Calculating the number of disc movements:
—see lecture—

7.8 Function guidelines

- (7.80) Ow To design good functions, you should know some rules, some of which are rendered here. A function may be called well designed if, among other respects, it is effective, clear

(algorithm quickly comprehensible) and easily maintainable (fixable and changeable).

(7.81) There are some **general guide lines** of how to build functions. Using these guide lines means improving the programming style. See also the short general remarks of how to build construct units (6.12).

- Functions should not be large (e. g. not more than about 100 lines), it is better to build smaller functions which call each other.
- Often a nested (cascading) design is useful:
 - functions for basic work,
 - functions for more complex activities that call the basic functions,
 - functions that call the more complex ones and perhaps the basic ones.

Advantage: when you build a function (write the function program text), you think about only a small problem area!

- Several kinds of functions should be distinguished:
 - functions with user interaction or input/output; if possible read and write in separated functions (e. g. read functions, write functions),
 - functions for performing calculations—these functions should not have any user interaction/input/output!! Otherwise these functions are not generally usable, e. g. when calculations with a value calculated elsewhere are wanted instead of calculations with an inputted value,
 - sometimes (seldom!) a mixture is necessary.

General rule: either user interaction or calculations!

- Interaction of a function should be done with the rest of the program only via the parameter list and return value of the function—very important: not via global variables. In contrast, global constants (and global types) are sensible, see also (7.53).
- No use of the `exit` function (`cstdlib` (12.82c), finishes the program at once), especially not in calculation functions. Reason: sometimes a function runs into an error, but how to recover cannot be decided here. Instead of finishing the complete program (e. g. CAD program), an error indicator should be set so that the calling location may decide what to do.

8 Storage classes, modular programming

8.0 Overview

This chapter renders the realization of the basic ideas presented language-independently in (ch. 6.3): storage classes and the programming style of modular programming.

Subchapter 1 describes the storage classes supported by C++ (storage class: different kinds of lifetime and data storage administration): static, automatic, and dynamic-controlled. The first two are presented here in more detail, the third will be explained later (ch. 10.3).

Subchapter 2 shows how to realize modular programming in C++, namely with the help of program files. Here, a good programming may be recognized by hiding as many names within a program file as possible, i. e. all names not directly needed from outside. Unfortunately, since this is done in C++/C only by an extra addition, many programmers often refrain from this very important hiding because of lack of thought. This can lead to poorly maintainable programs. Furthermore, the concept of header files is introduced to maintain consistency of ‘public’ names.

Sometimes, you have to link both C and C++ program files within one program. There are some specialities to be considered in this case (8.27).

8.1 Static and automatic storages class

(8.10) Ow As an expansion of the language-independent introduction (6.31), in this chapter the three storage classes supported by C++ are explained (8.11): the static, automatic, and dynamic-controlled storage class. The two following points show in more detail how the static and automatic storage classes are realized in C++. (8.14) renders a complete program example. It is essential for a programmer to know the properties and applications of these storage classes.

(8.11) A storage class defines the lifetime of a data memory space, the method of reserving and releasing. The language C++ directly supports three storage classes, the language C directly only two. General description of the first two storage classes see also (6.31).

- (a) Variables of the **static storage class** (statische Speicherklasse) ‘live’ during the complete program run time, their memory space is reserved once at the begin of the program run, it is released at the end of the program execution. They can get an initial value (‘initialization’), in general only from a constant expression (i. e. evaluable at compile time); if they are not explicitly initialized, they are initialized implicitly with their zero value in general.
- (b) Variables of the **automatic storage class** (automatische Speicherklasse) ‘live’ only during a block’s execution; their memory space is automatically reserved at entrance into the block and automatically released at exit. They may be initialized with an arbitrary expression; in general, there is no implicit initialization.
- (c) C++ The third **storage class dynamic-controlled** consists of storage spaces explicitly requested by the programmer with the operator `new`^{Op3k} and explicitly released with the operator `delete`^{Op3l}—absolutely independent of the program’s block structure. The area for these memory spaces is called free store (dynamic store, sometimes also name heap, details see (ch. 10.3)). In general, there is no implicit initialization.

↑↑ In C, there are library functions—mostly `malloc(size_t)` and `free()`—which serve to the same purpose (10.34).

Note With the ‘in general’ in the above initialization description of the three storage classes: The ‘in general’ may be broken by constructors (member functions to create and initialize objects), details see (9.21).

(8.12) **Static storage class**

Properties (see also (C++/5.2)):

lifetime	during complete program run time
initialization - explicit - implicit - how often	only with constant expression * [△] with their zero value * only once, mostly at begin of the program execution [△]

* Specialities with objects see (8.11 Note) and (9.21).

[△] ↑↑ Global variables are initialized once at the begin of program execution, static local variables once, too, but in C++ only when the program control flow encounters their definition for the first time. The initialization expression need not be constant in C++, it must be evaluable during run time when initializing.

Syntax for a variable’s definition with static storage class:

global variable	<i>DeclarationNormalForm</i> [▽]
local variable	¹³ static <i>DeclarationNormalForm</i>

[▽] Global variables always are static.

(8.13) Automatic storage class

Properties (see also (C++/5.2)):

lifetime	only during the block’s execution
initialization - explicit - implicit - how often	with an arbitrary expression NONE! * each time at the block’s entrance

* Specialities with objects see (8.11 Note) and (9.21).

Special case formal function parameter:

lifetime	only during the function block’s execution
initialization - how often	always, i. e. with the actual parameter each time the function is called

Syntax for variables with automatic storage class:

global variable	– not possible –
local variable	<i>DeclarationNormalForm</i> [△] [⊙]
Special case formal function parameter	¹⁴ <i>TypeSpecifier</i> _{1..n} ³⁰ <i>Declarator</i> [⊙]

↑↑ [△] It is permissible to put the storage class specifier ¹³auto before this, but it is left out normally without altering the meaning.

[⊙] If you put the storage class specifier ¹³register before this, the compiler is caused to put the variable or the formal parameter into a register if possible; this variable does not have a data memory address, so applying the address operator & (Op3g (10.21b)) is not allowed.

(8.14) Ex Stack, similar to (6.32):

```
#include <iostream>          // example program E08-14.CPP
using namespace std;

void errorPush();           // error message push
void errorPop();           // error message pop

int stack(bool pushYesNo,int value)
{
    static const int maxcnt=100;    // "static": static storage class
    static int storage[maxcnt],    // "static": static storage class
              count=0;

    if (pushYesNo) {
        if (count==maxcnt) errorPush(); // error
```

```

        else storage[count++]=value;
    }
    else {
        if (count)
            value=storage[--count];
        else {
            errorPop();                // error
            value=0;                    // to have a defined value
        }
    }
    return value;
}

void errorPush()
{ cout << "Max. number of stack elements reached!" << endl; }

void errorPop()
{ cout << "No elements available in stack!" << endl; }

int main()
{
    int i;

    cin >> i;
    stack(true,i);
    cout << stack(false,0) << endl; // value read
    cout << stack(false,0) << endl; // dummy value 0 - error before!
    for (i=0; i<100; ++i)
        stack(true,i);
    stack(true,-23);                // error
    cout << stack(false,0) << endl; // 99

    return 0;
}

```

8.2 Modular programming: splitting into several translation units

(8.20) Ov Modular programming is already described language-independently in (6.33). A **module** (in this sense) consists of several functions and data belonging to them. In C++, this is done with a program file: each program file represents a module.

In order to meet the demand for incapsulation (6.13), (8.21ff.) presents the possibilities of how to realize in C++ to open ('PUBLIC' in (6.34)) and to hide ('HIDDEN') functions as well as data. This may be done in C++/C by giving **internal linkage** to the names to be hidden; the public names must get **external linkage**. Since C++ is based on C, and C is relatively old, unfortunately hiding is not the 'normal case' which it should be; hiding is done only by an additional marking. Without this additional marking, the name has external linkage and is public, thus.

In C++(new), however, it is regarded as deprecated to hide names by using internal linkage. The new and preferable way to realize hiding is made with **unnamed namespaces** (8.23a). The general concept of the namespaces is briefly explained in (8.23c).

In order not to come into large consistency problems when handling public names, a very interesting concept has been realized already in C which is applied very often in C++, too: splitting off the declarations of names with external linkage into so-called **header files**. With the aid of an include directive, these header files are inserted into each program file in which the names appear, wherever they are used or they are defined. In (8.24) you find a detailed list of which parts belong to a header file and which ones to a program file.

Note *The type definitions, mostly as class definitions (ch. 9), are an exception; here even the definitions are generally put into header files.*

Two detailed C++ program examples (8.25, 8.26) explain how to handle several program files.

Specialities when linking C and C++ program files are summarized in (8.27).

(8.21)

- (a) To allow interaction between different translation units, the names within the object files (i. e. the compiled files) which are linked with each other must be reported to the linker. The locations where a reference is required (e. g. a function call) are linked (‘resolved’) by the linker with reference offers (e. g. a function definition). The names the linker gets from the object files have so-called **external linkage** (externe Bindung). Explanations of the terms object file, object program, and linker see (1.32).

In contrast to this, names of a translation unit which are hidden for the linker have so-called **internal linkage** (interne Bindung). Therefore, there cannot be any conflict between the same names in different translation units.

A better possibility to hide names is offered in C++(new) by an **unnamed namespace** (unbenannter Namensbereich). Normally, this will be used in the following.

- (b) Therefore, you should obey the following rules for interaction of different translation units:
- Normally, names should be hidden (by internal linkage or—better—by inclusion in an unnamed namespace), to be precise, because of clarity, then there are also no name collisions with other translation units.
 - Only if communication with other translation units is necessary, you should use external linkage.

This should be checked for each name! This rule also supports the hiding principle (6.13c).

- (c) Examples: names of functions which are called in other translation units must get external linkage. On the other hand, functions which execute service calculations for other functions of the same translation unit should be hidden. Variables should (almost) always be hidden!
- (d) In this context, the **one definition rule (ODR)** is important: Exactly one definition (reference offer) needs and must belong to external names, all other translation units are allowed to have only declarations of this name. This ODR is valid for internal names, too, of course.

↑↑ It is permissible according to the rules of C++/C for a linker not to distinguish between upper case and lower case letters or to regard only a limited number of characters in a name as distinctive. Today, this limitation is very seldom present.

You can omit a definition of a declared function if it is not called anywhere (7.23 Note3).

(8.22) External linkage

A global name gets external linkage:

function name	definition declaration	<i>NormalFunctionDefinition</i> extern_{opt} <i>NormalFunctionDeclaration</i>
variable name	definition declaration	<i>DefinitionGlobalVariableWithoutStorClassSpec</i> [*] extern <i>VariableDeclaration</i>
class name [⊙]	def./decl.	– always external linkage –
constant name	def./decl.	– see ↑↑2 –

* in general with initialization ⊙ s. (9.12b)

↑↑1 More precise rules see (Cpp/5.3).

↑↑2 Also constant names may have external linkage („**extern const** ...“, i. e. as definition, if initialized, else as declaration), but this is unusual. Instead, you may define e. g. the constant name with internal linkage (8.23) in each translation unit needed, i. e. in the header file (8.24c1) belonging to it.

- (8.23) In order to hide names within a program file, you may put them into an unnamed namespace (C++(new), see (a)), or give them internal linkage (older method in C/C++, see (b)).

- (a) C++(new)

In C++(new), there is a new and now preferable way to hide within a program file (i. e. in a module): the **unnamed namespace**.

The general concept of namespaces will briefly be explained in (c). Therefore, the unnamed namespace is introduced only formally without a detailed description of how it works: each declaration and each definition which should not be accessible outside a program file appears in such a namespace definition:

$${}^{40}\textit{Part\ Definition\ Unnamed\ Namespace} \doteq \text{namespace } \{ \textit{Declarations\ And\ Or\ Definitions} \}$$

It is unimportant how many such unnamed namespaces are built in a program file; the compiler regards all unnamed namespace definitions as being only one definition. For a function to be hidden, it is important that its declaration as well as its definition is put in such an unnamed namespace.

The effect is just the wanted one: if a name appears in such an unnamed namespace definition, this name is not accessible from other files. Thus, it is hidden in the sense of (6.32, 6.33) although—formally—it has external linkage. Within the same program file, you may have access to the name from everywhere, to be precise, from both locations outside and locations inside of an unnamed namespace.

You will see in (8.25) how to use an unnamed namespace.

(b) Internal linkage

A global name gets internal linkage:

function name	definition declaration	static <i>NormalFunctionDefinition</i> static <i>NormalFunctionDeclaration</i>
variable name	definition declaration [⊙]	static <i>DefinitionGlobalVariableWithoutStorClassSpec*</i> static <i>VariableDeclaration</i>
constant name	definition	const <i>DefinitionConstantWithInitializing</i> (without specifier extern) – cp. also (8.22 ↑↑2) –

* in general with initialization ⊙ seldom present

↑↑ More exact rules see (C++/5.3)

The example (8.26) demonstrates this application of **static**.

The key word **static** is used in different contexts with different meanings (it is a pity). It can mean static storage class as well as internal linkage. In order not to support this double meaning, you should use **static** only to mean static storage class and not any more to mean internal linkage. Therefore, in C++(new) the latter is regarded as deprecated (Str3/ch. B.2.3). Instead of this, you should use an unnamed namespace (a).

(c) ↑↑ Namespaces C++(new)

A namespace is a mean to group names logically. To avoid name conflicts, and not to overcharge the 'global' namespace (all names with external linkage from all program files), you may put a group of names into a namespace.

$${}^{40}\textit{Namespace\ Definition} \doteq \text{namespace } \textit{NamespaceName}_{opt} \{ {}^{10}\textit{Declaration}_{0..n} \}$$

In order to access names of a namespace, you may do so directly within this namespace, or outside by means of the explicit specification with the operator **::**^{Op1b}

$$\textit{NamespaceName} :: \textit{Identifier}$$

Instead of the explicit specification, all names of a namespace will be made accessible by a **using** directive, to be precise, for the scope in which the directive appears. (If there is a name conflict, the name of the scope prefers to that of the namespace; in this case you may use explicit specification.)

$$\textit{Using\ Directive} \doteq \text{using namespace } \textit{NamespaceName} ;$$

Recently, all names of the standard library are put into the namespace **std**, therefore the directive **'using namespace std;'**. If you use the compiler MS Visual C++ 6.0, the header files without **.h** put these names into **std**, those with **.h** into the global namespace.

When you omit *NamespaceName* in the namespace definition ('unnamed namespace', see (a)), the compiler generates an own unique namespace name, additionally a *UsingDirective* for this name. Since the programmer does not know this name, he or she cannot access names declared within it from outside the program unit. According (a), this property is used to hide names.

(8.24) Header files

- (a) It is very sensible to separate the declarations for names with external linkage from their definitions: header file (Headerdatei) and program file (Programmdatei). The former has mostly the file extension `.H` or `.h`, the latter mostly the extension `.CPP` or `.cpp`, also `.CXX` or `.cxx`.

The most important reason for this is the consistency between declaration and definition: since the programs ‘live’ in practice, i. e. are changed again and again, you may forget to adjust the declarations (possibly present at various locations) to match the definition if you change the name or the signature of a function. The linker does not always recognize this so that you may get errors which are hard to find. It is much better (and additionally accompanied by less effort) to place exactly one declaration into exactly one header file and to include this file anywhere where the declaration is required. It is very important, too, to include this header file also into the program file which includes the definition, so that the compiler can perform the consistency check!

- (b) Recommendations:

- If you only have few program files (practice: up to about five files), mostly it is sufficient to combine all declarations of all program files into a single header file and to include this file everywhere.
- Otherwise it is usual to create one header file for each program file (same file name, extension `.h` or `.H`); this header file is included wherever necessary.
- Some parts of a header file (e. g. class definitions (9.11)) must not be included more than once. If however a header file includes another header file (e. g. because of a required type definition), you can hardly prevent an indirect double inclusion. But with the possibility of conditional compilation, you can easily include an ‘include guard’; details see (5.75c) and the examples (8.25a) and (9.31a).
- Principle regarding the separation, detail see (c):
 - declarations needed in several program files, e. g. declarations of functions with external linkage, shall appear in exactly one header file,
 - the definitions of these functions must be done in exactly one program file,
 - however, functions in an unnamed namespace (or functions with internal linkage) – which shall be known only in exactly one program file – must not appear in a header file, the declaration and the definition have to be done in a program file.

- (c) Separation into header and program file:

- (1) Header file:

- inclusion of other (standard or own) header files—but because of clarity only those files directly needed in this header file
- declaration of functions with external linkage
- declaration of variables with external linkage
- following definitions as far as they are required **in more than one program file** (if—in contrast to this—only in one file: see (2)):
 - type definitions (**typedef**) (5.62)
 - classes (9.11)
 - macros (5.72, 5.73, 10.11b, 10.12b)
 - (if applicable) constants (10.11a)
 - **inline** functions (10.12a)
- Never: definition of an unnamed namespace (8.23b)
- Never: definition of functions (except **inline**)
- Please keep in mind: prevention of multiple inclusion with an include guard is sensible (5.75c)!

- (2) Program file:

- inclusion of the (standard or own) header files needed in this program file
- **important**: inclusion of the header file belonging to this program file (consistency check by the compiler!)
- declaration of functions and of variables with internal linkage
- definition of an unnamed namespace (8.23b)
- definition of functions

- definition of class members, e. g. member functions (as far as they are not `inline` (10.12a))
- definition of global variables
- following definitions as far as they are required **only in one program file** (if—in contrast to this—in several files: see (1)):
 - type definitions (`typedef`) (5.62)
 - macros (5.72, 5.73, 10.11b, 10.12b)
 - (if applicable) constants (10.11a)
 - `inline` functions (10.12a)



*Often the terms 'program file' and 'translation unit' are regarded as synonymous. The more precise definition: a **program file** is a file written by a programmer consisting of source text; with this, the preprocessor (ch. 5.7, essentially including the inclusions (`#include`) and expanding the macros (`#define`)) generates the **translation unit** which is offered to the compiler.*

(8.25) Example stack, similar to (6.34): three files: (a,b,c), here with the newer and preferable unnamed namespace. The same example will be rendered in (8.26) using the older way to hide names by internal linkage (`static`).

```
(a) // Example file E08-25A.H
#include <iostream> // prevents multiple inclusion
#define E08_25A_H_

// stack access functions:
void push(int value);
int pop();

#endif

(b) // Example file E08-25B.CPP
#include <iostream> // because of using cout in error... functions
using namespace std;

// declarations for this file: inclusion here for checking consistency!!
#include "e08-25a.h"

namespace { // unnamed namespace
    void errorPush();
    void errorPop();

    const int maxcnt=100;
    int storage[maxcnt],
        count=0;
}

void push(int value)
{
    if (count==maxcnt) errorPush(); // error
    else storage[count++]=value;
}

int pop()
{
    if (count) return storage[--count];

    errorPop(); // error
    return 0; // to have a defined value
}

namespace { // continue unnamed namespace
void errorPush()
{
    cout << "Max. number of stack elements reached!" << endl;
}
```

```

}}

namespace {
void errorPop()
{
    cout << "No elements available in stack!" << endl;
}
}

```

(c) // Example file E08-25C.CPP

```

#include <iostream>
using namespace std;
#include "e08-25a.h"

int main()
{
    int i;

    cin >> i;
    push(i);
    cout << pop() << endl;    // value read
    cout << pop() << endl;    // error; output dummy value 0
    for (i=0; i<100; ++i)
        push(i);
    push(-23);                // error
    cout << pop() << endl;    // 99

    return 0;
}

```

(8.26) The same example as in (8.25), to be precise, with the older way to hide names, namely by internal linkage (`static`).

(a) The header file E08-26A.H is identical to E08-25A.H with respect to the general sense, so it is not printed here.

(b) // Example file E08-26B.CPP

```

#include <iostream> // because of using cout in error... functions
using namespace std;

// declarations for this file: inclusion here for checking consistency!!
#include "e08-26a.h"

static void errorPush();    // "static": internal linkage
static void errorPop();    // "static": internal linkage

const int maxcnt=100;      // also internal linkage (8.23b)
static int storage[maxcnt], // "static": internal linkage;
        count=0;          // static storage class anyway (global)

void push(int value)
{
    if (count==maxcnt) errorPush(); // error
    else storage[count++]=value;
}

int pop()
{
    if (count) return storage[--count];

    errorPop(); // error
    return 0; // to have a defined value
}

static void errorPush() // "static": internal linkage
{ cout << "Max. number of stack elements reached!" << endl; }

```

```
static void errorPop()           // "static": internal linkage
{ cout << "No elements available in stack!" << endl; }
```

- (c) The main program file E08-26C.CPP is identical to E08-25C.CPP with respect to the general sense, so it is not printed here.

(8.27) ↑↑ **Common use of C++ and C (or other languages)**

Functions compiled in C must be declared for C++ as C functions because of different calling and naming conventions. Similarly, functions compiled within C++ can be called by C programs, too.

The meanings:

- *DeclarationCCppFunction*: declaration of a function either compiled in C++ and invoked in C or vice versa,
- *DefinitionCCppFunction*: definition of a function compiled in C++, but invoked in C (or in C++, too),
- *DefinitionCFunction*: definition of a function compiled in C, but invoked in C++ (or in C, too).

In any case, the property remains valid (and thus it will not be mentioned extra) that a function compiled in C++ can be invoked in C++ and similarly a function compiled in C can be invoked in C.

In C++ the header file looks like:

```
// C++ header file
extern "C" DeclarationCCppFunction
extern "C" {
    DeclarationCCppFunction1..n
}
// Note: the block above { ... } is called linkage block (Bindungsblock)
```

The C++ program file looks like (normal function definition):

```
// C++ program file
#include "C++ header file "
DefinitionCppFunction // as usual in C++
```

In C, you have to do this (header file):

```
// C header file
extern DeclarationCCppFunction // possible also without "extern"
// if necessary, several times: DeclarationCCppFunction
```

Note Here you have to pay attention to (7.23Note4) in any case, i. e. with parameterless function take `void` as parameter!

The C program file looks so:

```
// C program file
#include "C header file "
DefinitionCFunction // as usual in C
```

The supplements above are already contained for C++ in `<cstdio>` or the other C header files, too.

9 Object oriented programming: encapsulation of data and functions with access control

9.0 Overview

Details of the ideas for **object oriented programming** are—language-independently—described in (ch. 6.4). This chapter serves to transfer these ideas to C++.

Subchapter 1 shows how to generate and to use classes (in C++ as data types) and objects (variables of such a class type). Particular methods, namely constructors, are needed to generate an object at run time (subchapter 2). Thus, you may (for you as participant of this course: must) build constructors for sensible initialization. Subchapter 3 renders a detailed example, additionally, reasons are given for the principle of hiding data members of a class.

9.1 Classes and objects

- (9.10) Ov The purpose of a class is to encapsulate (put together) data and (access) functions with the possibility of explicit access control `HIDDEN` and `PUBLIC`. In C++ the class is a (data) type. This must be announced to the compiler with respect to its internal structure. (9.11) describes how to perform this class definition. Point (9.12) deals with the scope of class members and of the class name.

Variables of class type are called **objects**. Point (9.13) renders how to build and to use objects in C++.

(9.11)

(a)

The **class** is a **type** in C++, see (2.25a2, 3.20, 5.60, 5.61), i. e. a **user defined type**; the type is defined by the user. In contrast to the ‘built-in’ types—e. g. `int`, `double`—, the compiler does not know the internal structure of a class; therefore it must be informed about the structure. This is done by the **class definition**.

Note Other user defined types are e. g. arrays (ch. 5.4), because the user declares the component type and number of components.

^{10,11,12,14,21} *Class(Type)Definition* _[NC] \doteq

$$\text{class } \textit{ClassName} \{ \left[\begin{array}{l} \textit{AccessSpecifier} : \\ \textit{MemberDeclaration} \end{array} \right]_{0..n} \} ;$$

²²*AccessSpecifier* _[NC] \doteq `private` | `public`

Ex see (b).

↑↑ With the above class description, the compiler gets the complete information on how to construct variables of this type. Thus, this class description is actually a definition, cp. (7.21). This definition must be available to all translation units using the class, therefore it should appear in a header file (8.24c1). Despite the multiple existence of the class definition in the different translation units, the ODR (8.21d) is valid; identical definitions of the same class in different translation units are regarded as one single definition.

- (b) A class definition consists of:
- *ClassName* – it introduces the name of the class (type name).
 - sequence of *MemberDeclarations*; these members can be:
 - data members – syntactically similar to variable definitions,
 - member functions (functions which in general have to do something with the data members) – syntactically mostly function declarations,
 - additionally further declarations, for example types.
 - Optionally, arbitrarily often *AccessSpecifier*: (with colon), i. e. `private`: (then the following members are hidden) and `public`: (then they are accessible); if there is no specifier after the opening brace, so an implicit `private`: is set there.

Additionally, of course, the member functions—at least if they are used somewhere—must be defined; mostly this is done outside the class definition, i. e. in the program file belonging to this class.

Note1 A function definition inside a class definition is also allowed; in practice, this is done only with very small function blocks. In this case, the compiler regards the function as `inline` (10.12a, 11.11b).

Note2 Instead of the keyword `class`, the keyword `struct` is allowed, too, details see (11.11c).

Ex – see also (9.12 Ex, 9.13 Ex, 9.21 Ex) –

```
class Complex {
    double re, im;    // real part, imaginary part
public:
    // location for constructors, see point (21)
    void setCompl(double realP, double imagP);
    double real() { return re; } // decl. and definition of real()
    double imag();           // declaration von imag()
};
```

(9.12)

- (a) The **scope** (Gültigkeitsbereich) of a member is the area of the complete class (more exactly: starting from its declaration point), but however not outside the class. In order to open the class scope e. g. for definition of a member function (if it is done outside the class definition – this is normal!), you have to use the scope resolution operator `::` (**Op1b**):

ClassName::MemberName

The complete function block—additionally the parameter list, too—is regarded as belonging to the class scope; so you need not use the scope resolution operator there. Within this area you are allowed to access each class member—independently of an access restriction.

↑↑1 Also in a definition of member functions within the class definition (9.11b **Note1**), you may access all member names in the function block, also the ones following in the class definition.

↑↑2 It is permissible to use the scope resolution operator also within the function block (*ClassName::MemberName*); this is one of the possibilities to resolve name conflicts if applicable. Another similar possibility is using the **this** pointer (**this->MemberName**), see (11.13).

- (b) Class names always have **external linkage**, see (8.22).

Ex – from (9.11 Ex), see also (9.13 Ex, 9.21 Ex) –

```
// definition of the member functions of the class Complex:
void Complex::setCompl(double realP, double imagP)
{ re=realP; im=imagP; }

double Complex::imag()
{ return im; }
```

(9.13)

In C++, an **object** is a **variable of a class type**. Creation of such an object (such a variable) is done just as with variables of built-in types (e. g. with `int`), i. e. by a variable definition.

general variable definition: *Type VariableName ;*
special variable definition (object): *ClassName ObjectName ;*

Details about initialization of objects see (9.21).

In general, the access to a class’s member (data or function member) can be performed only via an object belonging to this class, i. e. via a variable of this class type: this access is done with the operator `.` (**Op2d**), as already explained in (5.23) for member functions:

ObjectName.MemberName

Note1 This is different with locations within the member functions themselves: these are allowed to access the members directly because they are implicitly linked with an object by the call.

Note2 About opening the class scope when defining member functions see (9.12a).

↑↑ There are also accesses without an object, i. e. with so-called static members. These will be explained later in (ch. 11.5).

Ex – from (9.11 Ex, 9.12 Ex), see also (9.21 Ex) –

```
Complex z;    z.setCompl(-1.7,4.2);
cout << z.real() << ' ' << z.imag() << endl;           // -1.7 4.2
```

9.2 Constructor and destructor

(9.20) Ow The introduction of constructors and destructors is a very great advantage of object orienting. Always when an object (i. e. a variable of class type) is generated, a special kind of member function is called, namely a constructor. In C++, it is impossible to generate an object without a constructor call. This presents the very large advantage that having sensible constructor definitions means that objects are always initialized.

The corresponding member function that is always called when an object is destroyed (or deleted) is the destructor. With this, you may define some clearing up. We will get to know reasonable applications of the destructor, beginning in (ch. 11.1).

(9.21)

(a) A **constructor** (Konstruktor) is a member function of a class which is automatically invoked when an object is created. Here the data members can—and should!!—be initialized appropriately so that an object does not contain uninitialized data in any case. This is really a **great advantage** over ‘normal’ variables.

A constructor is always invoked when an object is created. If this object has static storage class, it is called exactly once, if automatic storage class, it is invoked each time when entering the block because the object is newly created each time.

(b) The **constructor name** is the class name. A constructor does not have any return type (also not void), it must not have any **return** statement with a return expression.

In practice, mostly there are several overloaded constructors (7.61); distinguishing them is done by the signature. A constructor without a parameter is also called **default constructor** (Standardkonstruktor) because it is called when a variable is created without an (explicit) initialization value.

(c) If there is no constructor in a class definition, the compiler generates a default constructor which is empty (i. e. which does not do anything).

Ex – from (9.11 Ex, 9.12 Ex, 9.13 Ex) –

```
// within the class definition, at the point "location for constructors"
Complex() { re=im=0.0; } // decl. and def.
Complex(double realP, double imagP) { re=realP; im=imagP; } // decl. and def.

// usage elsewhere:
Complex c;    cout << c.real() << ' ' << c.imag() << endl; // 0 0
```

(9.22) Ex with detailed syntax for constructor calls:

```
// Example E09-22.CPP
#include <iostream>
using namespace std;
class Test {
    int x, y;
public:
    Test(); // default constructor (no par.)
    Test(int num); // constructor with 1 parameter
    Test(int numX,int numY); // constructor with 2 Parameters
    int getX(); // read x
    int getY(); // read y
```

```

};

Test::Test() { x=y=0; } // definition default constr.
Test::Test(int num) { x=y=num; } // definition constr. 1 par.
Test::Test(int numX,int numY) { x=numX; y=numY; } // def. c. 2 par.
int Test::getX() { return x; } // returns data member x
int Test::getY() { return y; } // returns data member y

int main()
{ // SYNTAX FOR CONSTRUCTOR CALLS according to number of parameters
  // (0): default constructor, (1): constr. 1 par., (2): constr. 2 par.
  // recommendation: first form in each line
  // implicit,      explicit,      implicit constructor call
  Test a;          Test b=Test(); // (0)
  Test c(4);       Test d=Test(4); Test e=4; // (1)
  Test f(1,3);     Test g=Test(1,3); // (2)

  Test h(3,4),i,j=2,k=Test(0,-1),l(3),m=Test(),n=Test(5);
  // VarName ConstrNumberPar: h 2; i 0; j 1; k 2; l 1; m 0; n 1

  cout << h.getX() << ' ' << m.getY() << ' ' << n.getY() << endl; // 3 0 5
  return 0;
}

```

(9.23)

- (a) A **destructor** (Destruktor) is a member function of a class which is automatically called when an object of this class is destroyed. Here, additional actions can be performed directly before releasing the memory space.

With a static object, destroying the object is done directly before program finishes, with an automatic object each time when exiting the block belonging to it.

- (b) The **destructor name** is formed from the tilde (~) and the class name. A destructor does not have—as a constructor—any return type (also not `void`), it does not have any parameters, it must not contain any `return` statement with a return expression.

↑↑ In contrast to constructors, a destructor cannot be overloaded because it must not have any parameters, so there is only one destructor for each class. If none is defined, the compiler generates an (empty) destructor by itself.

9.3 Examples

- (9.30) Ov The example (9.31) demonstrates how to build a stack with the help of object orienting. Point (9.32) describes with an example why the internal data structure of a class should be hidden. For students of this course this is a must: data members must not be public!

- (9.31) Ex Stack, similar to (6.42): three files (a,b,c)

```

(a) // Example file E09-31A.H
#ifdef E09_31A_H_ // prevents multiple inclusion
#define E09_31A_H_

class Stack {
public:
  Stack(); // default constructor
  ~Stack(); // destructor

  void push(int wert);
  int pop();

private:
  void errorPush();
  void errorPop();
}

```

```

        enum { maxcnt=100 };    // constant with class scope (12.21)
        int storage[maxcnt],
            count;
};
#endif

```

```

(b) // Example file E09-31B.CPP
#include <iostream>
using namespace std;
#include "e09-31a.h"           // definition of class Stack

Stack::Stack()               // default constructor
{
    count=0; // important initial value!!

    // Only for demonstrating the constructor - NOT FOR THE PRACTICE!
    cout << "Constructor call Stack" << endl;
}

Stack::~Stack()              // destructor
{
    // Only for demonstrating the destructor - NOT FOR THE PRACTICE!
    cout << "Destructor call Stack - "
         << "number of still existing elements: "
         << count << endl;
}

void Stack::push(int wert)
{
    if (count==maxcnt) errorPush(); // error
    else storage[count++]=wert;
}

int Stack::pop()
{
    if (count) return storage[--count];

    errorPop(); // error
    return 0; // to have a defined value
}

void Stack::errorPush()
{ cout << "Max. number of stack elements reached!" << endl; }

void Stack::errorPop()
{ cout << "No elements available in Stack!" << endl; }

```

```

(c) // Example file E09-31C.CPP
#include <iostream>
using namespace std;
#include "e09-31a.h"           // definition of class Stack

int main()
{
    Stack stackVar;
    int num;

    cin >> num;
    stackVar.push(num);
    cout << stackVar.pop() << endl; // value read
    cout << stackVar.pop() << endl; // error; output dummy value 0
    stackVar.push(-45);
}

```

```

    { // subordinate block: to demonstrate scope, constructor, destructor
      Stack stackVar;
      for (num=0; num<100; ++num)
        stackVar.push(num);
      stackVar.push(-23);          // error
      cout << stackVar.pop() << endl; // 99
    }

    cout << stackVar.pop() << endl; // -45
    return 0;
}

```

- (d) For detailed understanding, here the output of the above program—with comments not belonging to the output, each after the comment symbol `//`:

```

Constructor call Stack          // main() block
// input of number -935
-935                            // pop()
No elements available in Stack! // by errorPop()
0                               // pop(): dummy number
Constructor call Stack          // subordinate block
Max. number of stack elements reached! // by errorPush()
99                              // pop()
Destructor call Stack - number of still existing elements: 99 // subord.bl.
-45                             // pop()
Destructor call Stack - number of still existing elements: 0 // main() bl.

```

(9.32)

- (a) The distinction between `HIDDEN` and `ACCESSIBLE`, already mentioned in (6.13) and (6.42b) and described in more detail in this chapter, has an additional important advantage. If all data members are hidden and only access functions are public, then the implementation and with it also the internal representation of the data can be altered without the user knowing it, i. e. without the user having to alter his or her program code. Possibly only a new compilation is necessary.
- (b) An example for this is presented in the **exercises**; there the class `Date` is defined (here only a part):

```

class Date {
    int day;
    int month;
    int year;

public:
    Date();
    Date(int d,int m, int y);
    ~Date() {} // empty destructor

    int getDay() { return day; } // each line: def. of member function
    int getMonth() { return month; } // within the class definition,
    int getYear() { return year; } // see point (11b Anm.1) in this ch.

    // ...
};

```

An alteration of the internal representation does not require changing the interface. If the number of days between two date values of the same year must often be calculated (e. g. with the member function `difference(Date &otherDay)` in the example below or as a difference `date1-date2`), the date representation with `DayOfTheYear` is preferable to `Day` and `Month` (additionally `YearNumber`).

```

class Date {
    int dayYear; // instead of: int day; int month;

```

```

    int year;

public:
    Date();
    Date(int d,int m, int y);
    ~Date() {} // empty destructor

    int getDay(); // must be calculated now: def. member fct. elsewhere
    int getMonth(); // must be calculated now: def. member fct. elsewhere
    int getYear() { return year; }

    // now new function difference (number of days to otherDay):
    int difference(Date &otherDay);

    // ...
};

```

If in contrast date differences as number of days over even larger periods shall often be calculated (astronomy), the internal representation as the ‘Julian Date’ (JD, after Julius Caesar Scaliger, 16th century, Ital. naturalist and humanist): the days since Jan. 01, 4713 BC (zero point at 12:00 Universal Time) are counted — or the Modified Julian Date (MJD) with the zero point at November 17, 1858 at 0:00 UT. The time Jan. 01, 2000 0:00 UT is equivalent to 51 544 MJD or 2 451 544,5 JD.

(9.33) Further important possibilities of the object orientation are explained in (ch. 11).