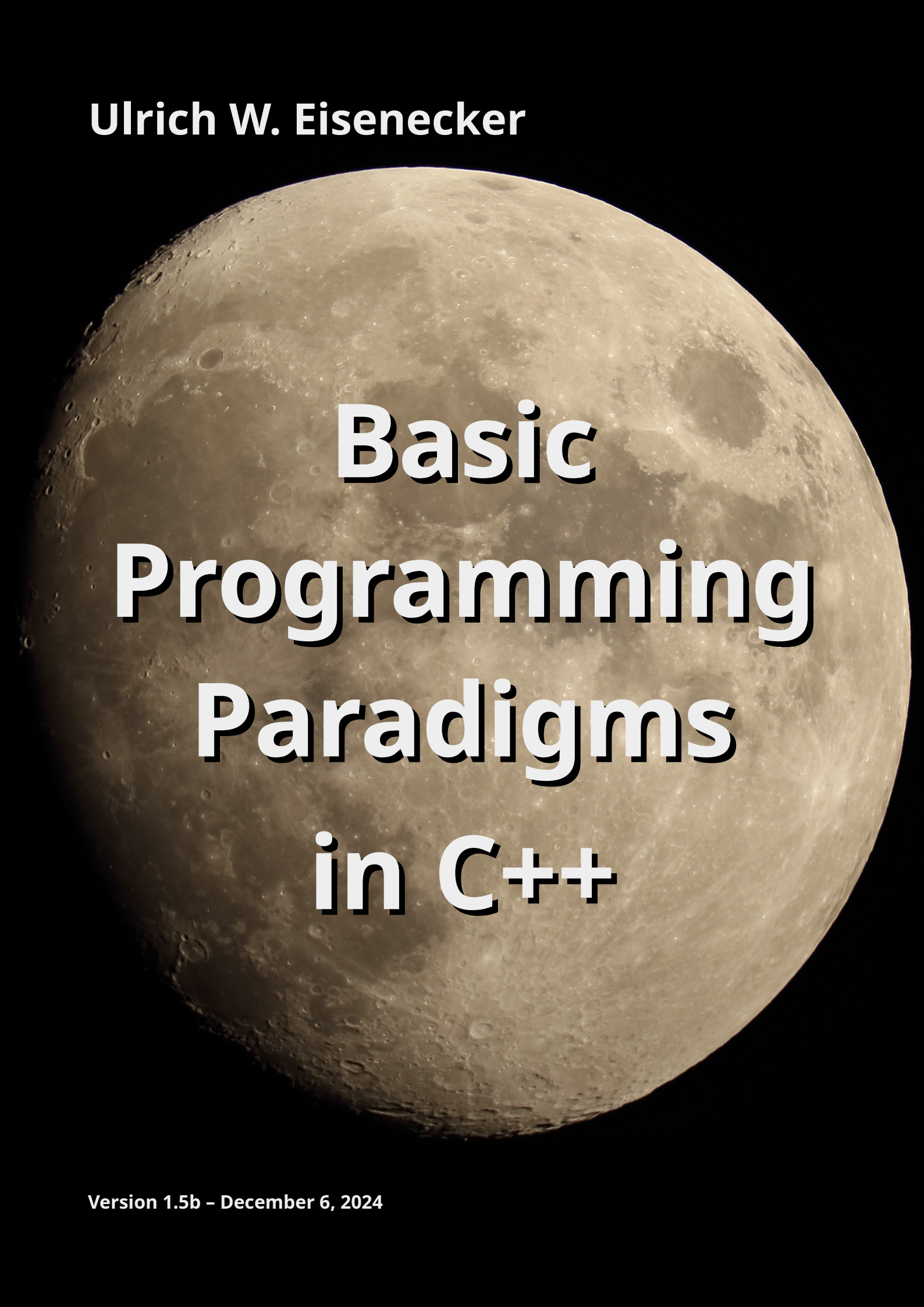


Ulrich W. Eisenecker



**Basic
Programming
Paradigms
in C++**

Version 1.5b - December 6, 2024

Author

Dr. Ulrich W. Eisenecker is a full professor of software development at the Institute of Information Systems at the Leipzig University.

Disclaimer

All information in this text has been written and compiled with care. Nevertheless, it may contain errors. Therefore, the author assumes no responsibility for the use of this text or the information contained therein.

The use of common names, trade names, product names, etc. in this text, even without special identification, does not justify the assumption that such names are to be considered free in the sense of the trademark and brand protection laws and may therefore be used by anyone.

If you feel that this work causes a legal issue that concerns you, please send an email to bpp-book@uni-leipzig.de with detailed information.

Cover Photo

The photo on the cover was taken by the author.

Download

The text and all accompanying material can be downloaded from <https://www.wifa.uni-leipzig.de/institut-fuer-wirtschaftsinformatik/professuren/professur-insbesondere-softwareentwicklung/studium/lehrveranstaltungen/bpp>. Newer versions will also be available for download from this website once they are released. Errata will also be published there.

License

Basic Programming Paradigms in C++ © 2022 – 2024 by Prof. Dr. Ulrich W. Eisenecker M.A. is licensed under **CC BY-SA 4.0**

Table of Contents

1. Preface.....	1
2. Background Information about Computing.....	3
2.1. Computing and Computers.....	3
2.2. A Simple Computer.....	3
2.2.1. First Task, First Program.....	3
2.2.2. Some Limitations and Risks.....	4
2.2.3. Second Task, Second Program.....	5
2.2.4. Generalized Program.....	5
2.2.5. Problem Space and Solution Space.....	6
2.2.6. Purposes of a Program.....	7
2.3. A New Computer.....	8
2.3.1. Porting the Program.....	8
2.3.2. Counting with Bits.....	9
2.3.3. An Infinite Design Space.....	10
2.3.4. Emulation of a Computer.....	11
2.3.5. Emulation vs. Simulation.....	12
2.3.6. Programming Paradigms.....	12
2.4. Afterthoughts.....	13
2.4.1. Human-Computer-System.....	13
2.4.2. Level of Detail.....	14
2.4.3. Program and Reality.....	14
2.4.4. Turing-Completeness.....	16
2.4.5. Abstract Machines.....	17
3. First Programs.....	18
3.1. Preparations.....	18
3.2. Values per Byte.....	20
3.2.1. Comments.....	20
3.2.2. Whitespace.....	20
3.2.3. #include Directive.....	21
3.2.4. Namespace.....	21
3.2.5. main() Function – First Information.....	21
3.2.6. Statement.....	22
3.2.7. return Statement.....	22
3.2.8. Wrap-Up.....	23
3.3. Values of Eight Information Units.....	23
3.3.1. Variables.....	24
3.3.2. Input.....	24
3.3.3. Ignoring Return Values.....	25
3.3.4. Line vs. Statement vs. Expression.....	25
3.3.5. Sequence.....	26

3.3.6. Wrap-Up.....	26
3.4. Values of Eight Validated Information Units.....	26
3.4.1. flush.....	27
3.4.2. Escape Sequences.....	27
3.4.3. ‘\n’ vs. endl.....	28
3.4.4. if Statement.....	28
3.4.5. Flowchart.....	30
3.4.6. Relational Operators.....	31
3.4.7. Standard Error Streams.....	31
3.4.8. Wrap-Up.....	31
3.5. Values of Information Units.....	31
3.5.1. At First Glance.....	32
3.5.2. Initialization of Variables.....	33
3.5.3. while Loop.....	34
3.5.4. Assignment Operator.....	34
3.5.5. Control Structures.....	35
3.5.6. Wrap-Up and Outlook.....	36
3.6. Functions.....	36
3.6.1. inputCardinality() Function.....	38
3.6.2. inputNumberOfInformationUnits() Function.....	39
3.6.3. calculateRepresentableValues() Function.....	39
3.6.4. outputRepresentableValues() Function.....	40
3.6.5. main() Function – More Information.....	41
3.7. Wrap-Up.....	42
3.8. Testing.....	42
3.8.1. Unit Tests.....	43
3.8.2. Regression Tests.....	46
3.8.3. Automated Unit Tests.....	47
3.9. Debugging.....	47
3.9.1. From Source Code to Executable Program.....	47
3.9.2. Preprocessor.....	49
3.9.3. Assembler.....	50
3.9.4. Linker.....	51
3.9.5. Example for Debugging.....	52
4. Further Details on Basic Concepts.....	60
4.1. Fundamental Types.....	60
4.1.1. Integral Types.....	61
4.1.1.1. bool Type.....	61
4.1.1.2. Character Types.....	65
4.1.1.3. Integer Types.....	65
4.1.1.4. std::byte Type.....	66
4.1.1.5. Arithmetic Operators.....	67

4.1.1.6. Promotions and Conversions.....	68
4.1.1.7. Assignment Operators.....	70
4.1.1.8. Floating Point Types.....	71
4.1.1.9. Real World and Computer – Again.....	77
4.2. More About Types.....	78
4.2.1. Information About Types.....	78
4.2.2. Pointers.....	83
4.2.3. Stack vs. Free Store.....	85
4.2.4. Dynamic Objects.....	86
4.2.5. References.....	88
4.2.6. Constants.....	90
4.2.6.1. Pointer to const.....	92
4.2.6.2. const Pointer.....	92
4.2.6.3. const Pointer to const.....	92
4.2.6.4. Naming.....	93
4.3. More About Functions.....	93
4.3.1. Declaration vs. Definition.....	96
4.3.2. Passing Parameters and Returning Results.....	98
4.3.2.1. Call by Value.....	98
4.3.2.2. Call by Pointer.....	100
4.3.2.3. Call by Reference.....	104
4.3.3. Returning Results.....	107
4.3.3.1. Return Type void.....	107
4.3.3.2. Returning a Reference.....	114
4.3.3.3. Returning a Value.....	115
4.3.3.4. Designing a Function.....	117
4.3.4. Parameters with Default Values.....	117
4.3.5. Function Overloading.....	118
4.3.6. Specifying Pointers, References, and Constness.....	123
4.3.7. Recursion.....	125
5. User-defined Types.....	126
5.1. Enumeration Types.....	126
5.2. Structured Datatypes.....	129
5.2.1. Rational Numbers With Fundamental Datatypes.....	130
5.2.2. Documentation Generation.....	135
5.2.3. User-defined Types for Related Data.....	139
5.2.3.1. Design Based on Reference Semantics.....	140
5.2.3.2. Design Based on Value Semantics.....	145
5.3. Abstract Data Types.....	150
5.3.1. Classes.....	151
5.3.2. Simple Class Buddy.....	152
5.3.3. Class Design Based on Reference Semantics.....	155

5.3.4. Class Design Based on Value Semantics.....	165
5.3.5. Reference- vs. Value-Based Design.....	169
5.4. Splitting of Programs.....	173
5.4.1. Header and Implementation Files.....	175
5.4.2. Include Guard.....	181
5.4.3. Preventing Name Collisions.....	182
5.4.4. Make.....	184
5.5. Overloading Operators.....	190
5.5.1. Motivation for Overloading Operators.....	190
5.5.2. The Syntax of Operator Overloading.....	192
5.5.3. Overloading Operators for RationalNumber.....	196
5.6. Testing.....	204
5.6.1. Motivation for Automated Testing.....	204
5.6.2. General Design of Tests.....	206
5.6.3. Testing the Mathematical Helper Function.....	207
5.6.4. Testing of RationalNumber.....	208
5.6.5. Testing Operators For RationalNumber.....	213
5.6.6. Full Test.....	215
5.6.7. Makefile And Testing.....	217
6. Motivation for the Case Study.....	220
6.1. A Rule-based Inference Engine.....	221
6.2. Requirements Analysis.....	223
6.2.1. System Documentation.....	224
6.2.2. Use of the Legacy System.....	232
6.3. Analysis Model.....	236
6.4. Design Model.....	239
6.4.1. Data.....	239
6.4.1.1. Digression.....	241
6.4.1.2. KnowledgeBase.....	242
6.4.1.3. LegalAnswers.....	243
6.4.1.4. Questions.....	246
6.4.1.5. Rule.....	247
6.4.1.6. Rules.....	250
6.4.1.7. Variables.....	252
6.4.1.8. Data Perspective Consolidated.....	253
6.4.1.9. Critical Review and Completion.....	254
6.4.2. Loading.....	258
6.4.3. Processing.....	266
6.5. Implementation.....	275
6.5.1. Include Files.....	275
6.5.2. Logger Class.....	276
6.5.3. toupper() Function.....	280

6.5.4. LegalAnswers Class.....	281
6.5.5. Variables Class.....	284
6.5.6. Pending Declarations.....	287
6.5.7. Questions Class.....	289
6.5.8. Rule Class.....	294
6.5.9. Rules Class.....	298
6.5.10. KnowledgeBase Class.....	299
6.5.10.1. KnowledgeBase Member Functions.....	301
6.5.10.2. Member Functions for Loading.....	301
6.5.10.3. Member Function for Testing.....	312
6.5.10.4. Member Function for Processing.....	312
6.5.10.5. Member Functions for Interaction.....	313
6.6. Evaluation.....	317
6.7. Outlook.....	318
7. References.....	321

List of Illustrations

Figure 1: Simple computer with stick and rings.....	3
Figure 2: Problem and solution space.....	7
Figure 3: Digital counter.....	8
Figure 4: Emulated stick-and-rings computer.....	11
Figure 5: Doll house (By diepuppenstubensammlerin from Ruhrgebiet Deutschland - 1974 OKWA dolls house, CC BY-SA 2.0, https://commons.wikimedia.org/w/index.php?curid=25946023).....	15
Figure 6: Source program in a plain text editor.....	18
Figure 7: Console-window.....	19
Figure 8: Source program in a browser window; previously “Run” was pressed.	19
Figure 9: Flowchart of the program shown in Listing 3.....	30
Figure 10: Abstract syntax tree for $a = (b + c) * 4$	48
Figure 11: Chained mappings of problem and solution spaces.....	49
Figure 12: Debugging Checksum.cpp with nemiver.....	54
Figure 13: Debugger shows function-call stack and local variables.....	56
Figure 14: Debugger shows source lines and assembler code.....	58
Figure 15: Sample dialog for executing FloatOutput.cpp (Listing 13).....	73
Figure 16: Output generated by for loop – exact computation (Listing 14).....	74
Figure 17: Output generated by for loop – inexact computation (Listing 15).....	74
Figure 18: Output of the Sizeof.cpp program (Listing 18).....	79
Figure 19: Type information in tabular form.....	81
Figure 20: Sample dialog for executing the CallByValue.cpp program.....	99
Figure 21: Sample dialog for executing the CallByPointer.cpp program.....	102
Figure 22: Variety of pointers.....	103
Figure 23: Output of the CallByReference.cpp program (Listing 46).....	105
Figure 24: Test of reverseString1() with “Hello”	111
Figure 25: Test of reverseString1() with “Hi”	111
Figure 26: Test of reverseString1() with “X”	111
Figure 27: index.html generated by Doxygen.....	137
Figure 28: File List generated by Doxygen.....	137
Figure 29: File reference generated by Doxygen.....	137
Figure 30: Call graph generated by Doxygen.....	138
Figure 31: Menu entry for classes.....	144
Figure 32: Documentation generated by Doxygen for struct RationalNumber...	144
Figure 33: Debugging the reference-based program, part 1.....	170
Figure 34: Debugging the reference-based program, part 2.....	171
Figure 35: Debugging the value-based program, part 1.....	172
Figure 36: Debugging the value-based program, part 2.....	173
Figure 37: Separate compilation, linking and execution.....	180
Figure 38: Dependency graph.....	184

Figure 39: Example directory structure for a C++ project.....	186
Figure 40: Report for successful test.....	215
Figure 41: Report for failed test.....	216
Figure 42: Focus on the solution space.....	220
Figure 43: Adaptive and unordered containers of the C++ STL.....	233
Figure 44: UML class diagram of the KnowledgeBase analysis model.....	237
Figure 45: KnowledgeBase class from the data perspective.....	242
Figure 46: LegalAnswers class from the data perspective.....	246
Figure 47: Questions class from the data perspective.....	247
Figure 48: Output for consulting ec/ORDERCND.KB with tracing enabled.....	248
Figure 49: Output for consulting ec/DUPLCND.KB with tracing enabled.....	249
Figure 50: Error message when consulting ec/CONTRCND.KB.....	249
Figure 51: Rule class from the data perspective.....	250
Figure 52: Rules class from the data perspective.....	251
Figure 53: Variables class from the data perspective.....	253
Figure 54: Consolidated class diagram from the data perspective.....	254
Figure 55: Processing of ec/TXT81.KB saved in UNIX format.....	256
Figure 56: Processing of ec/TXT81DOS.KB saved in MS-DOS format.....	257
Figure 57: Processing of ec/TXT81MAC.KB saved in macOS format.....	257
Figure 58: Processing of inputs by the ec/InputToken.cpp program.....	259
Figure 59: Interactive execution of the ec/InputToken.cpp program.....	260
Figure 60: Class diagram with integrated data and loading perspective.....	266
Figure 61: Sequence diagram for starting prove().....	268
Figure 62: Sequence diagram for further processing of prove().....	269
Figure 63: Flowchart for the Pseudo code of Rule::prove().....	271
Figure 64: Sequence diagram for the further processing of prove().....	272
Figure 65: Class diagram with integration of all perspectives.....	274
Figure 66: Program execution with logging enabled.....	280

List of Tables

Table 1: Comparison of programs for counting people and counting cups.....	5
Table 2: Programs for stick-and-rings computer vs. digital counter.....	9
Table 3: 16-bit memory.....	9
Table 4: 32,768 in big-endian and little-endian byte order.....	10
Table 5: Programs for stick-and-rings computer vs. emulated computer.....	11
Table 6: Abstract representation of the doll house in tabular form.....	16
Table 7: Schemes of if statements.....	29
Table 8: Expected results for testing calculateRepresentableValues().....	44
Table 9: Fundamental types in C++.....	61
Table 10: C++ string "Hello"s.....	108
Table 11: Algorithmic quantities for reverting a string (natural index values)...	109
Table 12: Natural index values for reverting "Hello"s.....	109
Table 13: Algorithmic quantities for reverting a string (C++ index values).....	109
Table 14: C++ index values for reverting "Hello"s.....	109
Table 15: Common German Grading System for Doctorates.....	126
Table 16: Classification of functions.....	174
Table 17: Evaluation of (a.add(b)).divide(a.subtract(c)); (1 st possibility).....	191
Table 18: Evaluation of (a.add(b)).divide(a.subtract(c)); (2 nd possibility).....	191
Table 19: Evaluation of (a.add(b)).divide(a_copy.subtract(c)) (1st possibility)....	191
Table 20: Evaluation of (a.add(b)).divide(a_copy.subtract(c)) (2 nd possibility)....	192
Table 21: Requirement candidates extracted from MAN and TUT.....	227
Table 22: Consolidated and structured requirements.....	232
Table 23: Further requirements for LEGALANSWERS.....	245
Table 24: Further requirements for QUESTION.....	247
Table 25: Further requirements for RULE and RULES.....	252
Table 26: Further requirements for variable names, text and Top Level.....	258
Table 27: Requirement 1.7.6 revised.....	292

List of Source Programs

Listing 1: ValuesPerByte.cpp.....	20
Listing 2: ValuesOfEightInformationUnits.cpp.....	24
Listing 3: ValuesOfEightValidatedInformationUnits.cpp.....	27
Listing 4: ValuesOfInformationUnits.cpp.....	32
Listing 5: ValuesOfInformationUnits.cpp with comment (excerpt).....	37
Listing 6: ValuesOfInformationUnitsWithFunctions.cpp.....	38
Listing 7: Blocks and hiding identifiers with the same name (excerpt).....	39
Listing 8: UnitTestOfCalculateRepresentableValues.cpp.....	45
Listing 9: Checksum.cpp.....	53
Listing 10: Booleans.cpp.....	63
Listing 11: UnsignedVsSigned.cpp.....	69
Listing 12: UnsignedVsSignedIntShorter.cpp.....	71
Listing 13: FloatOutput.cpp.....	73
Listing 14: for loop – exact computation.....	74
Listing 15: for loop – inexact computation.....	74
Listing 16: TestingFloats.cpp.....	75
Listing 17: Infinity_NaN.cpp.....	76
Listing 18: Sizeof.cpp.....	78
Listing 19: TypeInformation.cpp.....	80
Listing 20: NumericLimits.cpp.....	82
Listing 21: Pointer.....	84
Listing 22: Dereferencing a pointer.....	84
Listing 23: Dangling pointer.....	84
Listing 24: Checking for valid pointer.....	85
Listing 25: Incompatible pointer types.....	85
Listing 26: Dynamic object.....	86
Listing 27: Dereferencing a pointer.....	86
Listing 28: Using a dereferenced pointer.....	86
Listing 29: Releasing a pointer.....	87
Listing 30: References.cpp.....	88
Listing 31: ReferenceToPointer.cpp.....	89
Listing 32: Euler’s number.....	90
Listing 33: Euler’s number as constant.....	90
Listing 34: Reference to const for const object.....	90
Listing 35: Reference to const for non-const object.....	90
Listing 36: PointerAndConst.cpp.....	91
Listing 37: ChecksumMain.cpp.....	94
Listing 38: ChecksumFunction.cpp.....	95
Listing 39: checksum() with [[nodiscard]] attribute.....	96
Listing 40: Declaration, which is also a definition.....	97

Listing 41: Declaration and separate definition.....	97
Listing 42: CallByValue.cpp.....	99
Listing 43: Pass call-by-value parameter as const.....	100
Listing 44: CallByPointer.cpp.....	101
Listing 45: const pointer as function parameter.....	104
Listing 46: CallByReference.cpp.....	105
Listing 47: integralDivision() function with reference-to-const parameters.....	106
Listing 48: Alternative declaration of integralDivision() function.....	106
Listing 49: ReturnVoid.cpp.....	110
Listing 50: Paper-pencil test of reverseString1() function for “X”.....	112
Listing 51: ReturnVoidImproved.cpp.....	113
Listing 52: ReturnReference.cpp.....	114
Listing 53: ReturnValue.cpp.....	115
Listing 54: ReturnValue2ndVersion.cpp.....	116
Listing 55: Prototype of the estimatedRange() function.....	117
Listing 56: Prototype of estimatedRange() function with default parameters....	117
Listing 57: Overloading.cpp.....	119
Listing 58: MoreOverloading_A.cpp.....	120
Listing 59: MoreOverloading_B.cpp.....	121
Listing 60: MoreOverloading_C.cpp.....	122
Listing 61: AsteriskAndAmpersand.cpp.....	123
Listing 62: * and & placed immediately before the variable name.....	124
Listing 63: & and & placed immediately after the type.....	124
Listing 64: EffectOfConst.cpp.....	125
Listing 65: Grades.cpp.....	127
Listing 66: RationalNumberSimple.cpp.....	133
Listing 67: RationalNumberSimple_Doxyfile.....	136
Listing 68: struct RationalNumber.....	139
Listing 69: Declaration of a variable of RationalNumber type.....	139
Listing 70: Sending RationalNumber data members to cout.....	139
Listing 71: Default initialization of data members in a struct.....	140
Listing 72: RationalNumberStructureReferenceSemantics.cpp.....	143
Listing 73: Alternative design of add() function.....	143
Listing 74: RationalNumberStructureReferenceSemantics_Doxyfile.....	143
Listing 75: RationalNumberStructureValueSemantics.cpp.....	148
Listing 76: inputInt() function.....	148
Listing 77: Declaration and initialization of variables by function call.....	148
Listing 78: Combining normalization and initialization.....	148
Listing 79: normalize() function rewritten.....	149
Listing 80: InputRationalNumber() function rewritten.....	150
Listing 81: add() function rewritten.....	150
Listing 82: Schema for defining a class.....	151

Listing 83: Buddy.cpp.....	153
Listing 84: Passing a Buddy exemplar by value.....	154
Listing 85: RationalNumberClassReferenceSemantics.cpp.....	159
Listing 86: One calculation per statement.....	162
Listing 87: An expression that combines all calculations.....	163
Listing 88: Multiplying a RationalNumber variable by itself.....	163
Listing 89: Dividing different RationalNumbers gives the correct result.....	163
Listing 90: Dividing RationalNumber by itself gives wrong results.....	163
Listing 91: Improved implementation of RationalNumber::divide().....	164
Listing 92: Changing the declaration of RationalNumber::divide().....	164
Listing 93: RationalNumberClassValueSemantics.cpp.....	168
Listing 94: RatNumRefSem_1/math_helper.hpp.....	175
Listing 95: RatNumRefSem_1/math_helper.cpp.....	176
Listing 96: RatNumRefSem_1/rational_number.hpp.....	177
Listing 97: RatNumRefSem_1/rational_number.cpp.....	179
Listing 98: RatNumRefSem_2/math_helper.hpp.....	181
Listing 99: RatNumRefSem_2/rational_number.hpp.....	182
Listing 100: RatNumRefSem_3/math_helper.hpp.....	182
Listing 101: RatNumRefSem_3/math_helper.cpp.....	183
Listing 102: RatNumRefSem_3/rational_number.hpp (abridged).....	183
Listing 103: RatNumRefSem_3/rational_number.cpp (excerpt).....	183
Listing 104: RatNumRefSem_3/main.cpp (excerpt).....	184
Listing 105: RatNumRefSem/makefile.simple.....	187
Listing 106: RatNumRefSem/makefile.....	189
Listing 107: RatNumRefSem/Doxyfile.....	189
Listing 108: Evaluation of (a.add(b)).divide(a_copy.subtract(c));.....	191
Listing 109: Overloading operator +() as a member function.....	193
Listing 110: Overloading operator +() as a free function.....	194
Listing 111: RatNumRefSemOp/rational_number.hpp (excerpt).....	197
Listing 112: RatNumRefSemOp/rational_number.cpp (excerpt).....	198
Listing 113: RatNumRefSemOp/rational_number_operators.hpp.....	198
Listing 114: RatNumRefSemOp/rational_number_operators.cpp.....	199
Listing 115: RatNumRefSemOp/main.cpp.....	201
Listing 116: RatNumValSemOp/rational_number.hpp.....	203
Listing 117: RatNumValSemOp/rational_number.cpp.....	203
Listing 118: Declaring and inputting a RationalNumber (reference-based design).....	204
Listing 119: Declaring and inputting a RationalNumber (value-based design).....	204
Listing 120: RatNumRefSemOp/src/full_test.cpp.....	207
Listing 121: RatNumRefSemOp/src/math_helper_test.cpp.....	207
Listing 122: RatNumRefSemOp/src/rational_number_test.cpp.....	211
Listing 123: RatNumRefSemOp/src/rational_number_operators_test.cpp.....	214
Listing 124: RatNumRefSemOp/makefile.....	218

Listing 125: RatNumRefSemOp/Doxyfile.....	219
Listing 126: GOAL rule from UN_AD_CN.KB.....	234
Listing 127: LEGALANSWERS rule from UN_AD_CN.KB.....	234
Listing 128: ANSWER rule from UN_AD_CN.KB.....	234
Listing 129: Exemplary QUESTION rule from UN_AD_CN.KB.....	234
Listing 130: Exemplary IF rule from UN_AD_CN.KB.....	234
Listing 131: Complete UN_AD_CN.KB knowledge base.....	235
Listing 132: ec/51SameLegalAnswers.cpp.....	243
Listing 133: ec/LA1.KB.....	244
Listing 134: ec/LA0.KB.....	244
Listing 135: ec/101SameQuestions.cpp.....	246
Listing 136: ec/ORDERCND.KB.....	248
Listing 137: Rule of ec/ORDERCND.KB with reverse order of conditions.....	248
Listing 138: ec/DUPLCND.KB.....	248
Listing 139: ec/CONTRCND.KB.....	249
Listing 140: ec/401RuleLines.cpp.....	251
Listing 141: ec/VAR41.KB.....	255
Listing 142: ec/VAL41.KB.....	255
Listing 143: ec/TXT81.KB.....	256
Listing 144: ec/InputToken.cpp.....	259
Listing 145: ec/Input2TokenTypes.cpp.....	261
Listing 146: Pseudo code for loading a knowledge base.....	263
Listing 147: Pseudo code for inputting a LegalAnswer.....	264
Listing 148: Pseudo code for inputting a Rule.....	265
Listing 149: Pseudo code for inputting a Question.....	265
Listing 150: Declaration of KnowledgeBase::prove().....	267
Listing 151: Declaration of Rules::prove().....	267
Listing 152: Pseudo code of Rules::prove().....	268
Listing 153: Declaration of Rule::prove().....	269
Listing 154: Pseudo code of Rule::prove().....	269
Listing 155: Declaration of KnowledgeBase::askValue().....	272
Listing 156: Declaration of Questions::ask().....	273
Listing 157: Pseudo code of Questions::ask().....	273
Listing 158: EC.cpp – included files.....	275
Listing 159: EC.cpp – Logger class.....	277
Listing 160: ec/LoggerDemo.cpp (excerpt).....	279
Listing 161: EC.cpp – toupper() function.....	280
Listing 162: EC.cpp – LegalAnswers class.....	282
Listing 163: EC.cpp – Variables class.....	285
Listing 164: PingPongImpossible.cpp.....	288
Listing 165: PingPong.cpp.....	289
Listing 166: EC.cpp – Questions class.....	291

Listing 167: Questions::add(), which fully complies with requirement 1.7.6.....	291
Listing 168: ec/EC.cpp – Rule class.....	295
Listing 169: EC.cpp – Rules class.....	299
Listing 170: EC.cpp – Data members of KnowledgeBase class.....	300
Listing 171: EC.cpp – KnowledgeBase::error().....	302
Listing 172: Checking for and reporting an error.....	302
Listing 173: Checking for and reporting an error with KnowledgeBase::error().....	302
Listing 174: EC.cpp – KnowledgeBase::inputToken().....	303
Listing 175: EC.cpp – KnowledgeBase::inputIsAre().....	304
Listing 176: EC.cpp – KnowledgeBase::inputVariableValue().....	305
Listing 177: EC.cpp – KnowledgeBase::inputLegalAnswers().....	306
Listing 178: EC.cpp – KnowledgeBase::inputGoal().....	307
Listing 179: EC.cpp – KnowledgeBase::inputRule().....	307
Listing 180: ANIMAL knowledge base (excerpt).....	308
Listing 181: EC.cpp – KnowledgeBase::inputQuestion().....	309
Listing 182: EC.cpp – KnowledgeBase::inputAnswer().....	310
Listing 183: EC.cpp – KnowledgeBase::input().....	310
Listing 184: EC.cpp – KnowledgeBase::output().....	312
Listing 185: EC.cpp – KnowledgeBase::prove().....	312
Listing 186: EC.cpp – KnowledgeBase::inputCommand().....	313
Listing 187: EC.cpp – KnowledgeBase::run().....	314
Listing 188: EC.cpp – various.....	315
Listing 189: EC.cpp – main().....	316

1. Preface

The target audience for this text is students of *computer science* or *information systems*. It is the companion book to the *Basic Programming Paradigms* lecture. It introduces procedural programming and programming with abstract data types and gives a first insight into object-oriented programming. The rule-based paradigm is also presented, but as an application; it is not used for programming.

In addition to pure programming, other programming-related activities are presented, namely (automated) documentation, (automated) testing, requirements engineering, and analysis and design. Selected characteristics of software quality are also discussed.

The text also addresses computational thinking as presented in (Abelson & Kong, 2019). This includes the essential areas of data practices, modeling and simulation practices, problem-solving practices, and systems thinking practices. In particular, computational thinking is used to understand aspects of programming or to develop solutions to problems in programming.

C++ is used as a means to pursue the aforementioned goals. The aim of this text is not to provide a comprehensive overview of the C++ language and its libraries. Instead, a didactic approach was chosen in which basic paradigms and concepts for programming a solution are presented using small applications. Finally, the programming of an expert system shell serves as a realistic case study. To emphasize the use of C++, the title of this text has been changed to *Basic Programming Paradigms in C++*.

Platforms and tools are also important. Programming only in theory is mostly pointless. But there are so many platforms, i.e. operating systems or web browsers, and tools that it is impossible to present them even in extracts. To look for tools and platforms and to try them out depends decisively on oneself.

Nevertheless, here are some recommendations. For the first steps in programming, an editor and a compiler running in a web browser may be sufficient. Very soon, more will be needed. *Ubuntu*, a *Linux* distribution, and an additionally installed *GNU compiler collection* including a C++ compiler, be it *g++* or *clang*, are well suited for the next programming steps. For editing program texts, *Vim* or *Gvim* (*mvim* on *macOS*) is a useful tool, available both as a console application and as an application with a graphical user interface. In some cases, an integrated development environment (*IDE* in short), such as *Code::Blocks*, can be useful because it provides an integrated visual debugger. Otherwise, it is sufficient to use command line tools such as *Doxygen*, *make*, etc. Learning command line tools pays off for a lifetime. The investment in learning a graphical user interface is very often lost as soon as a new version of the operating system or the corresponding application is released.

The *LibreOffice* suite was used for writing this text. *Google Noto* fonts were used for the typesetting. For programming, the tools mentioned above were used.

The text was originally written in English by a non-native speaker. To check the linguistic quality www.deepl.com was very helpful.

This text is intended for online reading, e.g. on a tablet computer. It does not have a book layout. If a printed version is desired, a color printer should be used. This is also one reason why there is no keyword index in the appendix. Instead, the electronic version is simply searched for the desired keyword or its stem.

I would like to thank Uli Breyman, Nico Willert, Christoph Diesener, Iuliia Shcherbina, Janik Eriksson, Tom Marvin Schmiedke, Fritz Böhme, Valentin Morio, Hoang Nguyen, Yannis Hübenthal and other students whose names escape me for reading parts of earlier versions of this text and pointing out errors.

I thank Lennard Apel, who redrew most of the illustrations.

I am indebted to my wife, who patiently endured the many hours I spent with the computer to create this text and all the programs, and not with her.

If you would like to suggest changes or improvements, report errors, or you want to communicate any other concern related to this text please provide detailed information about the text passages and any necessary supporting or background information. Please e-mail them to bpp-book@uni-leipzig.de.

2. Background Information about Computing

2.1. Computing and Computers

Computing originates in the Latin verb *computare*, which means *to calculate* or *to estimate*. Basically, computing means to perform computations using some device. Today, *computer* refers to an *electronic device* performing the computation. A computing device must reveal the result of the computation or, formulated more abstractly, the result must be measurable, for example, by looking at it. Optionally, data for performing the computation may be entered. The computation is described by a program. The program includes instructions that control its execution and data that is processed during execution.

2.2. A Simple Computer

Imagine a stick with a plate at its lower end and rings with holes which can be slipped over the stick. Together, the stick and the rings constitute a primitive computing device (Figure 1).

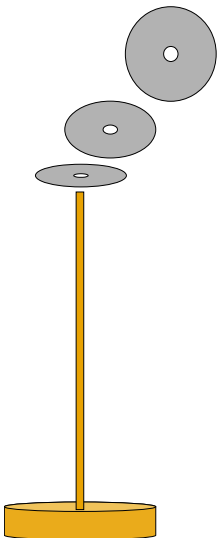


Figure 1: Simple computer with stick and rings

2.2.1. First Task, First Program

Imagine, there is a house and you have to find out, how many people are in the house. Here is the program for performing this task:

1. Before entering the house remove all rings from the stick and get a reserve of rings.
2. Enter the first room of the house.
3. For each person you see, slip one ring over the stick.
4. If there is a next room go to it and repeat the previous step.
5. Leave the house and count the number of rings slipped over the stick. The result equals the number of persons in the house.

It has to be mentioned that, if the condition of the fourth step does not apply, the program is continued with the next step. The computing device consists of the stick and the rings. The result of the computation is always visible and easy to measure. It equals the number of rings slipped over the stick, thus counting them is sufficient. There are two operations for entering data, namely removing a ring and slipping a ring over the stick. Before starting a computation, all rings have to be removed from the stick, after completing a computation the result can be obtained by counting the number of rings on the stick.

This computer does not do anything what you cannot do yourself as a human being. But it is patient. If the computation is interrupted it can be continued any time later. Especially, if you forgot – for whatever reason – the number of people already counted, this computer does remember it.

2.2.2. Some Limitations and Risks

Obviously, there are some limitations and some risks using this computer.

A first limitation is, that using the computer relies on your ability to recognize different people in a room and to remember for which of them you already slipped a ring over the stick. A second limitation is that people may neither enter or leave the house nor move between rooms as long as the computation is in progress. To forbid moving between rooms is necessary because of the first limitation, which requires to memorize only people that have been counted in one room but not in the entire house. A third limitation is that all rooms must be accessible and you may not omit a room.

Now for the risks. The first risk is that you point the stick – for whatever reason – downwards and the rings fall from the stick. This risk is relevant when a computation is in progress. After the result has been obtained, it is no longer relevant, when rings are removed from the stick. The second risk is, that the stick breaks. This may prevent a computation or stop it prematurely. A third risk is, that the supply of rings runs out, which may prevent the completion of a computation.

2.2.3. Second Task, Second Program

Let us consider another task, namely counting the number of cups in a kitchen. Cups may be stored in various cupboards, in the dish washer, and somewhere else in the kitchen. The goal is to write a program for accomplishing this task using the same level of abstraction as for the previous program counting people in a house.

Maybe the result is a program similar to this:

1. Before entering the kitchen remove all rings from the stick and get a reserve of rings.
2. Examine the first place in the kitchen.
3. For each cup you see slip one ring over the stick.
4. If there is a next place examine it and repeat the previous step.
5. Leave the kitchen and count the number of rings slipped over the stick. The result equals the number of cups in the kitchen.

Table 1 places the two descriptions side by side, whereby the differences are colorized.

Counting people	Counting cups
<ol style="list-style-type: none"> 1. Before entering the house remove all rings from the stick and get a reserve of rings. 2. Enter the first room of the house. 3. For each person you see, slip one ring over the stick. 4. If there is a next room enter it and repeat the previous step. 5. Leave the house and count the number of rings slipped over the stick. The result equals the number of persons in the house. 	<ol style="list-style-type: none"> 1. Before entering the kitchen remove all rings from the stick and get a reserve of rings. 2. Examine the first place in the kitchen. 3. For each cup you see slip one ring over the stick. 4. If there is a next place examine it and repeat the previous step. 5. Leave the kitchen and count the number of rings slipped over the stick. The result equals the number of cups in the kitchen.

Table 1: Comparison of programs for counting people and counting cups

2.2.4. Generalized Program

Probably, there is a large number of similar programs, all of them counting items in a structured space. Writing all these programs would be a tedious affair. Is it possible to devise a program focusing on the commonalities while abstracting from differences? Yes, it is. To do so, a few generalizations are required:

- “house”, “kitchen” → “space”
- “enter”, “examine” → “visit”
- “room”, “place” → “sub space”
- “of”, “in” → “of”
- “person”, “cup” → “item”

Applying these generalizations gives the following program (the generalized terms are colorized):

1. Before entering the **space** remove all rings from the stick and get a reserve of rings.
2. **Examine** the first **sub space** of the **space**.
3. For each **item** you see slip one ring over the stick.
4. If there is a next **sub space** **visit** it and repeat the previous step.
5. Leave the **space** and count the number of rings slipped over the stick. The result equals the number of **items** in the **space**.

The more general this resulting program is, the more generally applicable it is. The amount of its generality depends on how successfully task-specific details were identified and replaced by adequate abstractions.

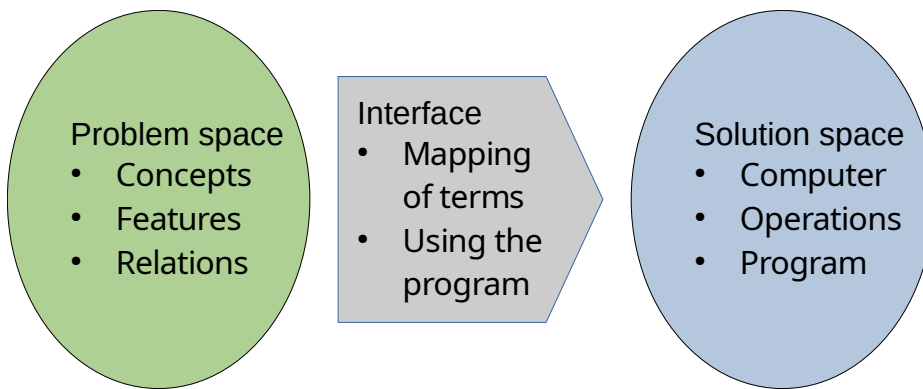
Obviously, this program can be used for, e.g., counting the pieces of cutlery in a drawer. Before doing so, the user of the program has to consider that *space* means *drawer*, *sub space* means *compartment*, *item* means *piece of cutlery*. For this reason, the user has to have a sufficient understanding not only of the specific task, but also of the program. Besides the necessary mapping of general terms to task-specific vocabulary the user must also know how to handle the computing device, that is, the stick and the rings as well as the operations slipping and removing a ring and counting the final number of rings to obtain the result.

2.2.5. Problem Space and Solution Space

Interestingly, this offers a first opportunity to distinguish between the user’s perspective of a program and the perspective of a programmer, who writes the program.

For the user it is essential to understand the task they should accomplish, while the programmer must know the computing device, the computing operations, and the structure of the program. These two perspectives, they may be adequately called ***problem space*** and ***solution space***, are interfaced by the program (Figure 2). From the user’s perspective using a program should only require minimal additional knowledge beyond their specific task.

Figure 2: Problem and solution space



Thus, using the generalized program requires some program-related knowledge, namely the mapping of task-specific vocabulary to the possibly more abstract terms of the program. A program, specifically designed for a certain task, requires less knowledge.

Obviously, this is a tension between two poles. ***The writing of task-specific programs is a burden for the programmer, while using a generalized program is a burden for the user. The programmer's burden increases with each additional task-specific program they have to write, while the user's effort in learning a more generalized program amortizes with each use of the program.***

For now, it is sufficient to be aware of this tradeoff and to understand its consequences. Of course, this tension can be mitigated, but for now this will not be done.

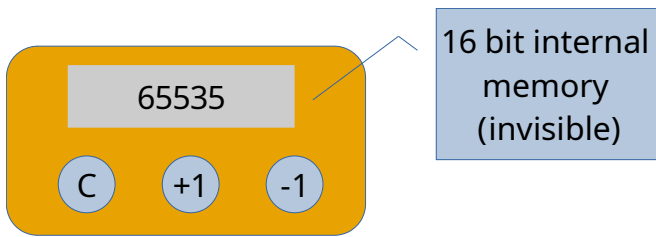
Knowing about problem and solution space, the relation between them, the purposes of a program, and the roles of a user and a programmer are absolutely essential in software development.

2.2.6. Purposes of a Program

The preceding paragraph mentions the *purposes* of a program. Why is there more than one purpose? Of course, the first purpose of a program is to support the user in performing a task. But there is a second purpose as well. The program also serves the programmer. A programmer must read and understand a program

- to fit it to a changing task,
- to create a copy of it which is adapted to a new, but similar task afterwards,
- to increase its generality,
- to port it to a new computer,
- to fix errors.

If a program makes these activities difficult or prevents them, it can suddenly become unusable if a task changes slightly or if one task is replaced by another.



Moreover, the program cannot be used to support structurally equivalent tasks or serve as a template for creating programs for new but similar tasks.

2.3. A New Computer

For whatever reason, one day the stick-and-rings-computer is replaced by an electronic device. It will be called *digital counter* (Figure 3). Its case has a display which always shows the state of its internal memory interpreted as an unsigned integral number. It offers three buttons triggering the operations *clear*, *increment by one*, and *decrement by one*. Accordingly, the buttons are labelled *C*, *+1*, and *-1*. *-1* is provided for convenience only, in case *+1* has been pressed inadvertently. Then *+1* can be instantaneously redone without having to press *C* to start anew. In fact, *-1* is not covered explicitly in the following program. The internal memory has a capacity of 16 bit. A *bit* is the smallest information unit and will be explained in detail soon.

Figure 3: Digital counter

2.3.1. Porting the Program

Now, the program for the stick-and-rings computer has to be ported to the digital counter. The generalized program for counting items in a structured space will serve as a starting point. The specifics of the stick-and-rings computer are mapped to the digital counter:

- “remove all rings from the stick” → “press *C*”
- “and get a reserve of rings” → (nothing) – the internal memory must suffice
- “slip one ring over the stick” → “press *+1*”
- “count the number of rings slipped over the stick” → “read the number on the display”

For comparison, Table 2 aligns both versions side-by-side with all changes highlighted.

Stick-and-rings computer	Digital counter
1. Before entering the space remove all rings from the stick and get a	1. Before entering the space press <i>C</i> .

Stick-and-rings computer	Digital counter
<p>reserve of rings.</p> <ol style="list-style-type: none"> Examine the first sub space of the space. For each item you see slip one ring over the stick. If there is a next sub space visit it and repeat the previous step. Leave the space and count the number of rings slipped over the stick. The result equals the number of items in the space. 	<ol style="list-style-type: none"> Examine the first sub space of the space. For each item you see press +1. If there is a next sub space visit it and repeat the previous step. Leave the space and read the number on the display. The result equals the number of items in the space.

Table 2: Programs for stick-and-rings computer vs. digital counter

2.3.2. Counting with Bits

No knowledge of the digital counter's internal operation is required to use it. It is sufficient to know about the effect of pressing a certain button. Nevertheless, the following program illustrates a possibility how to count using bits.

The memory consists of 16 switches which correspond to the individual bits. If a switch is open, this means a bit with value 0, if a switch is closed, this means a bit with value 1. To depict the memory the bits are ordered from right to left from the lowest to the highest bit. Table 3 gives an impression with all bits set to 0. The lowest bit has rank 0, the highest bit has rank 15. The bit with the lowest rank is also called the *least significant bit* (abbr. *LSB*), and the bit with the highest rank is called *most significant bit* (abbr. *MSB*)

	↓ Highest bit															Lowest bit ↓
Rank	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 3: 16-bit memory

Now, a program can be formulated which adds 1 to any number being representable by this memory:

- Define a pointer and let it point to the lowest bit.
- If the bit the pointer points to is 0, change it to 1 and stop the program.
- Change the bit, to which the pointer points, to 0.
- If the pointer already points to the highest bit, issue an error message and stop the program.

5. Let the pointer point to the next higher rank and continue with step 2.

Well, this program does not clarify everything. For example, it is not described how the pointer is realized. However, it illustrates how the immediate successor of an integral number is computed. As a consequence, it becomes obvious that there are 2^{16} possible different patterns of 0's and 1's which can be mapped to integral numbers from 0 to 65,535.

Knowing this one may come up with programs for setting all bits to 0 and for decreasing a number representable by this memory by 1. This can be done as an exercise.

2.3.3. An Infinite Design Space

There is one more important aspect. The realization of the memory described above is only one out of a large number of possibilities. For example, a closed switch could represent a bit with value 0 and an open switch could represent a bit with value 1. The bits could be ordered ascending from left to right from lowest bit to highest bit. Bits could be grouped in bytes, each byte comprising 8 bits. Thus, a memory with 16 bits would consist of 2 bytes.

These two bytes may be arranged in two ways: High Byte first or Low Byte first. This byte order, which is called *endianness*, is very important, because it can vary between different computer systems. The byte order with the highest byte first and the lowest byte last is called **big-endian**. The byte order with the lowest byte first and the highest byte last is called **little-endian**. Table 4 shows how the integer 32,768 is represented in big-endian and little-endian byte order. In the following, a big-endian byte order is assumed unless otherwise specified.

Big endian	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Little endian	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

Table 4: 32,768 in big-endian and little-endian byte order

Besides these purely structural variations anything else, that can be physically implemented, could be used for representing the two different states of a bit. And, by the way, the constraint of using bits is not a necessary one. Also, decimal numbers or numbers of any numeral base could be used.

Obviously, the only relevant aspect for writing a program is the abstract definition of a computer and its operations. The inner structure of a computer and how its operations are implemented should not be relevant for the programmer. Of course, for a person designing a computer and implementing it this is essential to know.

2.3.4. Emulation of a Computer

Assume, the stick-and-rings computer should be replaced by a digital counter in such a way, that the user does not experience a substantial change. *Substantial change* means, the corresponding program is altered fundamentally. To avoid this simply the housing and the display of the digital counter are exchanged. Instead an integer the display shows a stick with rings. Furthermore, only two of the three buttons are used. They are labelled *add ring* and *remove ring*. Figure 4 shows how the resulting computer could look.

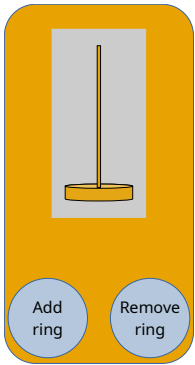


Figure 4: Emulated stick-and-rings computer

Stick-and-rings computer	Emulated computer
<ol style="list-style-type: none"> 1. Before entering the space remove all rings from the stick and get a reserve of rings. 2. Examine the first sub space of the space. 3. For each item you see slip one ring over the stick. 4. If there is a next sub space visit it and repeat the previous step. 5. Leave the space and count the number of rings slipped over the stick. The result equals the number of items in the space. 	<ol style="list-style-type: none"> 1. Before entering the space remove all rings from the stick. 2. Examine the first sub space of the space. 3. For each item you see slip one ring over the stick. 4. If there is a next sub space visit it and repeat the previous step. 5. Leave the space and count the number of rings slipped over the stick. The result equals the number of items in the space.

Table 5: Programs for stick-and-rings computer vs. emulated computer

This requires only little changes when rewriting the program. Both versions are compared in Table 5 using two columns and highlighting the changes.

The only change is, that “*and get a reserve of rings*” is removed. This is definitively no problem. It may be possible to slip about 10 to 20 rings over the stick, so they still can be discriminated visually. This value is considerably below the limit of 65,535.

It is a fundamental insight that one computer can be replaced by another computer in a way, that the replacing computer emulates the replaced computer. This means, that the emulation mimics the original to an extent, so that the emulation is almost indistinguishable from the original – at least, not from the perspective of a program.

As emphasized before, a program is written against an abstract specification of a computer and its operations. A consequence of emulation is, that an abstract specification of a computer and its operations can be implemented by another program, provided both abstract specifications and operations are powerful enough. Eventually, an abstract specification of a computer and its operations must be implemented in hardware. But this implementation may be very different from the original program. This way the solution space introduced before can be – and in practice most often is – just another problem space, namely that of programming language design – which eventually maps to a solution space which is related to the underlying hardware.

2.3.5. Emulation vs. Simulation

There is an important difference between *emulation* and *simulation*. An emulation is a full replacement of the original, while the simulation contains important aspects of the original but cannot replace it. Emulating or simulating a coffee machine may result in the insight, that it takes 5 minutes to produce a pitcher of fresh coffee. But the emulator produces real coffee one can drink, the simulator does not.

2.3.6. Programming Paradigms

The background of performing computation is now almost complete. In the following text a computer is introduced which is programmed in a programming language called C++. The choice of C++ is arbitrarily. C++ encompasses many ideas how to program a computer. Such an idea, for example, *object-oriented programming*, is called a *programming paradigm*. A programming paradigm combines specific concepts, their features and their relations. It is even possible to combine concepts of different programming paradigms. This may lead to conflicting or emergent phenomena. Various programming paradigms and their interactions will be presented in the following.

2.4. Afterthoughts

2.4.1. Human-Computer-System

A critical look at the first program reveals that it does not exclusively address the stick-and-rings computer. In the following version a comment at the end of each line denotes the addressee of this part of the program. Each comment starts with `//` and is highlighted as well.

1. Before entering the house remove all rings from the stick and get a reserve of rings. `// This step addresses the user; removing the rings involves the computer.`
2. Enter the first room of the house. `// This step addresses the user.`
3. For each person you see, slip one ring over the stick. `// This step addresses the user; slipping a ring involves the computer.`
4. If there is a next room, go to it and repeat the previous step. `// This step addresses the user.`
5. Leave the house and count the number of rings slipped over the stick. The result equals the number of persons in the house. `// This step addresses the user; counting the rings involves the computer.`

All steps are directed to the user, not to the computer! Two steps do not involve the computer at all (2 and 4). In three steps the user makes use of the computer (1, 3 and 5). There is no step exclusively directed to the computer.

Obviously, this program is more for a user, but not for a computer. This is remarkable, but not non-sensical. Even more typical programs involve a user. The user provides input, evaluates the state of the computer, and possibly their observation influences their decision-making, thus altering the next input of the user.

Thus, it is appropriate to consider user, task, and computer as a system of its own, a so-called *human-computer system*. Despite the task is not mentioned in this compound term, it is relevant as well. A user can perform a task without using a computer. ***Introducing a computer will and usually should change how the task is executed*** and it should have an impact on qualities like efficiency and efficacy.

The preceding program sketching counting with bits addresses only one actor. This actor may be a human user or a computer. Thus, it is an example for a program which could be executed exclusively by a computer.

2.4.2. Level of Detail

The steps of the first program are not very consistent with respect to their level of detail. *Level of detail* means that part of the used vocabulary is well-defined, but the rest is not. Relatively well-defined are *stick* and *ring* as objects and *slip* and *remove* as well as *remove all* as operations. The terms *house*, *room*, *person*, *see*, *go to next room*, *enter house*, and *leave house* have no explicit definition at all. Their use relies exclusively on the conception a presumed user has in mind.

A program for a computer may only use vocabulary which is known to the computer, that is, either it is part of the programming language used to instruct the computer, or precise definitions must be provided elsewhere, for example, being part of the program itself or residing in a *library*, which is accessible to the program. A library can be considered as a collection of reusable program units for a specific purpose.

2.4.3. Program and Reality

As outlined before, humans carry out tasks in real life – without a computer. If a computer is used for executing the task it becomes also part of reality. A program running on the computer usually refers to some part of the reality outside the computer, but in the context of the task. In the first program, this is the number of people in a house. In the digital-counter computer, this number is represented using a memory of 16 bits.

16 bits cannot represent an arbitrary large number, but in most practical cases 16 bit are sufficient. If the computer had to explore the house by itself, the house and its various rooms including people, must be represented in the computer as well. In this case the programmer has to decide which data exactly and with which structure it has to be represented. This representation has to keep track of the rooms in the house, maybe how these rooms are connected to each other, and about the number of people in the rooms. Probably it would not record the physical dimensions of the house and its rooms or weight and age of people in a room. Hence, this representation is an *abstraction* of reality. It *simulates* a certain aspect of reality, but it does not *emulate* reality.

Figure 5 shows the photograph of a doll house. It is only a two-dimensional color-image of a model of a true house with rooms and people in it. But it gives an idea of a real house.

Figure 5: Doll house (By diepuppenstubensammlerin from Ruhrgebiet Deutschland - 1974 OKWA dolls house, CC BY-SA 2.0, <https://commons.wikimedia.org/w/index.php?curid=25946023>)



Table 6 presents a possible abstract representation of it, which consists of 11 integers, comprising 8 bits each. The first number represents the number of rooms of the house. It does not discern between various kinds of rooms, such as normal room, staircase, balcony, or gallery. Each following number represents the number of people in a room. It does not differentiate between adults and children or sex.

This representation is relatively flexible. The format of an integral number is fixed and the first integral number tells how many integral numbers follow.

Description	Representation
Number of rooms (including 2 staircases, 2 balconies, 1 gallery)	0000 1010
People in room 1 (lower left)	0000 0000
People in room 2 (lower right)	0000 0000
People in room 3 (stair between rooms 1 and 3)	0000 0001
People in room 4 (middle left)	0000 0001
People in room 5 (middle right)	0000 0010
People in room 6 (right balcony)	0000 0000
People in room 7 (left balcony)	0000 0100

Description	Representation
People in room 8 (upper left)	0000 0000
People in room 9 (stair between room 5 and gallery)	0000 0000
People in room 10 (gallery)	0000 0000

Table 6: Abstract representation of the doll house in tabular form

2.4.4. Turing-Completeness

Alan Turing wrote a fundamental paper about *computable numbers* (Turing, 1937). A computable number can be computed by a *computable function*. A computable function defines how to compute that number. The program for adding 1 to an integral number described in Section [Porting the Program](#), is an example of a computable function. Another example is the well-known definition of the *factorial* of an integer n :

$$\begin{aligned} factorial(n) &= n \cdot factorial(n-1) \\ factorial(0) &= 1 \end{aligned}$$

As an example, Turing proved that any function of an integral variable with a recursive definition is a computable function. This also applies to functions which are defined by using computable functions. The preceding function *factorial* is an example of a function with a recursive definition, or in short, a ***recursive function***. A recursive function makes use of itself in its definition. In the first part of the above definition, the factorial function calls itself to compute values above 0. The second part is a specialization for the value 0, which also terminates the recursion.

Furthermore, Turing conceptualized a *universal machine* being able to compute any computable function. Roughly, this machine corresponds to a *tape* which begins at the left and extends endless to the right. It contains *programming instructions* and *data* encoded in some *alphabet*. Another essential component is a *read/write head* which can be positioned arbitrarily on that tape. The instructions on the tape refer to the operations *read*, *write*, and *moving* the read/write head. According to these instructions the read/write head is moved across the tape, reads data from or writes data to the tape. This way, data or instructions already present on the tape, can be altered, that is, they can be overwritten or deleted.

This universal machine cannot be built in reality, because it requires an endless tape. Furthermore, it could take endless time processing a tape. **Hence, a real computer can compute either only a limited range of numbers accurately or it can approximate computable as well as other numbers with limited accuracy and range.** Doing so, may require a long, but finite time.

Today, a set of operations which can represent computable functions is said to be *Turing-complete*. In fact, such a set of operations plus the rules to combine these operations constitute the language in which a computer can be programmed. From a formal point of view, all programming languages, which are Turing-complete, have the same expressive power. That means, each Turing-complete programming language can be used to represent any computable function – within the practical restrictions mentioned before.

Both, the *Universal Turing Machine* and *Turing-completeness*, have fundamental implications for programming:

1. ***Any computer of the class of computers being as powerful as a Universal Turing Machine can emulate any other computer of this class.***
2. ***Any problem with a computable solution can be represented using any Turing-complete programming language.***

It has to be kept in mind, that the phrase *problem with a computable solution* is in practice generally restricted by limitations of space and time.

2.4.5. Abstract Machines

As it was already introduced in Section [An Infinite Design Space](#), an *abstract machine* defines an idealized computer in terms of its operations, its inputs, and its outputs. As such, a Universal Turing Machine is an abstract machine. One possible implementation of an abstract machine is a real computer, that is, hardware. Another possibility is its implementation as a program which is executed by another abstract machine. This way, an abstract machine emulates another abstract machine, which has been mentioned before in Section [Emulation of a Computer](#).

In reality, a program which runs on an abstract machine must be eventually executed on a hardware computer. This way, a large gap between problem space and solution space can be bridged by stacking abstract machines, each representing a manageable step from problem to solution space.

Basically, a programming language can be considered as an abstract machine. From this point of view, C++ is a computer whose operations and rules for combining them correspond to the programming language C++.

For those, more deeply interested, (Diehl et al., 2000) provides an overview of abstract machines for implementing programming languages.

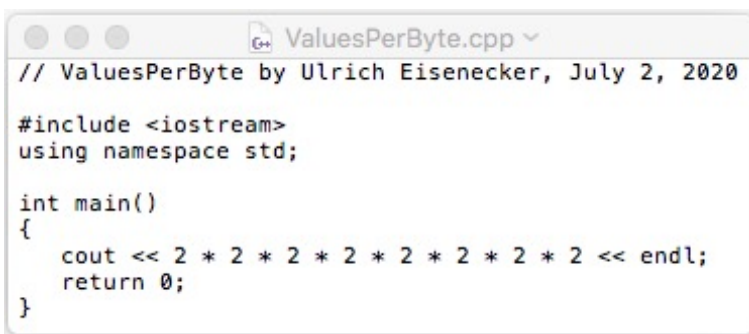
3. First Programs

3.1. Preparations

Writing and executing a program requires the following steps:

1. Editing the program text, that is, writing and formatting it.
2. Compiling the program text into an executable program.
3. Executing the program.

The first step requires a text editor which can produce plain text. Plain text means, the text contains no additional formatting information and it is saved as pure ASCII characters. Figure 6 shows a source program in a plain-text editor.



```
ValuesPerByte.cpp
// ValuesPerByte by Ulrich Eisenecker, July 2, 2020

#include <iostream>
using namespace std;

int main()
{
    cout << 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 << endl;
    return 0;
}
```

Figure 6: Source program in a plain text editor

Different platforms provide different editors for this purpose. For example, *macOS* provides *TextEdit*, *Ubuntu gedit*, and *Microsoft Windows notepad.exe*. How to use these editors is not explained here. Furthermore, there are many more editors especially designed for editing programs, either stand-alone or being integrated in a development environment. An example of a powerful, portable, open source editor is *Vim* (<https://www.vim.org>). For editing the first programs one of the simple editors mentioned before will suffice.

As a compiler *g++* is a good choice. It is open source and available for many platforms. It is the responsibility of the readers to inform themselves how to install it on their computer, and to actually install it.

A console window is required to execute the binary program. The operating system of the computer will provide at least one program opening a console window. Again, the readers should inform themselves which console window is available and how to operate it. Figure 7 shows a console window on a Unix-based operating system. Running the *ls* command displays the contents of the working directory. Then, the command *g++* is executed to compile the source program *ValuesPerByte.cpp* to the binary program *a.out*. After that, the contents of the directory are displayed again

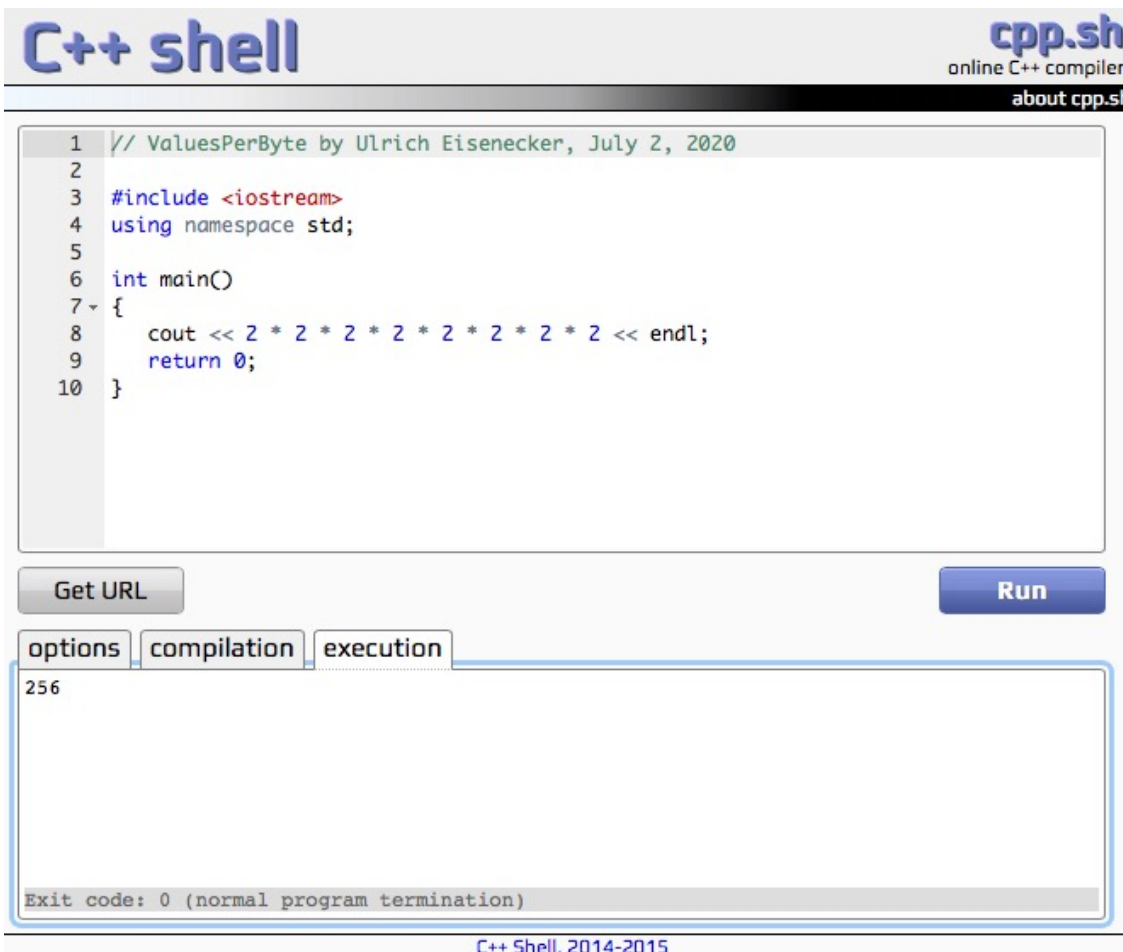


```
sources — -bash — 79x8
Ulrichs-Mini:sources ulrich$ ls
ValuesPerByte.cpp
Ulrichs-Mini:sources ulrich$ g++ ValuesPerByte.cpp
Ulrichs-Mini:sources ulrich$ ls
ValuesPerByte.cpp      a.out
Ulrichs-Mini:sources ulrich$ ./a.out
256
```

with `ls`. Finally, the binary program `a.out` is executed by entering `./a.out` and pressing the *Return* key. The dot in `./a.out` refers to the current directory that contains `a.out`, and the slash separates the directory from the name of the program.

Figure 7: Console-window

Another option is to search the internet for a C++ compiler which runs in a web browser. There are various web-sites which allow to enter program text and compile/execute it immediately. This covers all the three steps mentioned above and may be the easiest option for the start (Figure 8).



C++ shell cpp.sh
online C++ compiler
[about cpp.sh](#)

```
1 // ValuesPerByte by Ulrich Eisenecker, July 2, 2020
2
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     cout << 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 << endl;
9     return 0;
10 }
```

Get URL Run

options | compilation | execution

```
256
```

Exit code: 0 (normal program termination)

[C++ Shell, 2014-2015](#)

Figure 8: Source program in a browser window; previously “Run” was pressed

Both possibilities are illustrated above. Subsequently, these steps will be addressed explicitly only for a specific purpose.

3.2. Values per Byte

A *bit* is the smallest and as such the elementary information unit. It has only two values, either 0 or 1. A *byte* is a larger information unit which comprises 8 bits. The program in Listing 1 computes, how many different values can be represented using 1 byte. It performs the necessary computations and outputs the result in a console window.

```
1 // ValuesPerByte by Ulrich Eisenecker, July 2, 2020
2
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     cout << 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 << endl;
9     return 0;
10 }
```

Listing 1: ValuesPerByte.cpp

3.2.1. Comments

The first line of Listing 1, `// ValuesPerByte by Ulrich Eisenecker, July 2, 2020`, is a *comment*. A comment is not processed by the compiler. It contains either information for the programmer or other programs that are able to process special comments (more on this later). The first form of a comment in C++ begins with `//` and stops at the end of the line. The second form of a comment in C++ begins with `/*` and stops with the next matching `*/`. This form of comment can be embedded in a single line or extend over several lines. Comments in C++ cannot be nested.

3.2.2. Whitespace

Lines 2 and 5 of Listing 1 are empty lines. An empty line has no effect. Empty lines help to structure a source program visually, making it easier to read and to comprehend for the programmer.

Empty lines, spaces, and tabs are examples of the so-called *whitespace*. Whitespace is used to separate elements of a C++ program and improve its presentation for the human reader. It is not processed by the compiler.

3.2.3. #include Directive

Line 3 of Listing 1, `#include <iostream>`, includes another file with program text required to use a library for input and output.

C++ consists of a *language core* and libraries. The source program of a library is written in C++. A library does not extend the C++ programming language, but it provides new functionality for a selected purpose. There are many libraries which are defined in the C++ standard as well as a wealth of non-standard libraries. The C++ standard is defined by the *International Organization for Standardization*. The current standard is informally referred to as C++20.

Each line starting with the character `#` (after optional whitespace) is processed by a special tool, the *preprocessor*. The preprocessor runs before the compiler and its output is fed to the compiler. The `#include` directive includes another text file as if it were an immediate part of the including text file.

From C++20 onwards, the inclusion of header files for the use of libraries is replaced by the import of modules. As modules are currently not sufficiently supported by compilers and libraries have not been converted into modules, modules are not introduced in this text, but the inclusion of header files is used throughout.

The need for the preprocessor will be eliminated by the upcoming C++ standards. Hence, it will be explained only up to the extent which is required to understand the programs presented in this introduction.

3.2.4. Namespace

Everything provided by a library of C++ is wrapped in a so-called *namespace*. Namespaces are provisions for avoiding different definitions of the same name. For standard libraries the namespace `std` is used. Some standard libraries make even use of special namespaces nested in the namespace `std`.

The line using `namespace std;` imports everything of the namespace `std`, so that it can be used without additional measures subsequently. This line corresponds to a *statement* in the C++ language. As such it must be terminated with a semicolon.

3.2.5. main() Function – First Information

In line 6 of Listing 1 `int main()` starts the definition of a *function* called `main()`. Functions are a central concept in C++. A function is the smallest encapsulated unit of a program. It optionally accepts *parameters*, performs something, e.g., a computation, and optionally returns a result. While `main` is the name of the function, the subsequent pair of matching parentheses embraces the parameters

which are passed to the function. There are no parameters here, that is, this function has no parameters. The function does return a result, which is a signed integer. This is indicated by the preceding `int`, which is the name of a built-in type of the C++ programming language. It is a convention to refer to a function by its name followed by a pair of parentheses, for example, `main()`.

The implementation of a function is contained in a *block*. A block begins with an opening curly brace, `{`, (line 7 in Listing 1) and it ends with a closing curly brace, `}`, (line 10 in Listing 1).

3.2.6. Statement

The line `cout << 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 << endl;` (line 8 in Listing 1) contains a *statement*. As such, it is terminated with a semicolon. Here, the statement contains several operations. First, `2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2` is examined. This is an *expression* with seven occurrences of the operator `*`, which represents multiplication. The multiplication *operator* takes two numerical values as *operands* and computes the result. Evaluating the first occurrence of `2 * 2` gives the value 4, which is subsequently multiplied with 2. The last operation is to multiply 128 with 2, resulting in the value 256. Rewriting the entire line using this computed value yields `cout << 256 << endl;`. `cout` is a so-called *object*. It represents the *standard output*. The standard output is a *stream* of characters which is usually tied to a *console window*. `cout` is the left operand of the `<<` operator. This operator sends a textual representation of its right operand to the output stream. Here, the right operand is an integer value. Thus, executing this line has the effect of outputting 256 in the console window. Interestingly, the output operator, also known as *stream-insertion operator*, returns its left operand as a result of its execution, that is, executing `cout << 256` returns `cout`. Hence, the next occurrence of the `<<` operator sends again a textual representation of its right operand to `cout`. In this case, the right operand is an object named `endl`. This is a special object. It is a predefined so-called *manipulator* which starts a new line in the output stream. For performance reasons, streams are usually associated with *buffers*. `endl` additionally forces any buffered output to be sent immediately to the output stream. This way, the text eventually appears in the console window.

3.2.7. return Statement

The next statement is `return 0;` (line 9 in Listing 1). A return statement completes a function. If the function promises to return a value, a return statement must specify a value of the designated return type. Here, the `main()` function returns 0, which means that it executed successfully. Return values of `main()` larger than 0 indicate errors, but it is not defined, which error is associated with which value. Returning a

value indicating an error is a special convention for the `main()` function. It does not apply to functions in general. It is possible to omit the return statement in `main()`. In this case, `main()` automatically returns 0 on exit. This option only applies to the `main()` function.

A function returning the type `void` returns nothing. Such a function is exited either explicitly by `return;` or implicitly if there is no more statement to be executed.

A function may have more than one exit. But it is considered a good programming style to keep the number of exits small. Ideally, there is only one exit. In this way, the function is easier to read, understand and analyze.

3.2.8. Wrap-Up

The first program shows some features of the programming language in which it has been written. These features have been introduced only superficially. More detail will be added later.

Nevertheless, this program performs a computation and outputs its result. The computation itself is rather simple and does not reveal what it is about. By looking exclusively at `2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2` it is almost impossible to tell what is computed. That is, because the program does not convey any meaning of the result 256. The meaning has to be told separately to the user, either in an accompanying documentation of the program, by choosing a concise name for the program, or by outputting the result with an explanation.

In practice, this program is of very limited use. Its result can be easily computed by humans without additional means, and it cannot be used to compute results for similar problems. Nevertheless, it has the advantage that it makes the computation explicit and persistent. If necessary, it would be easy to fix an error, re-compile, and re-run the program. It would also be simple to add another eight multiplications by 2, thus obtaining the values that can be represented using two bytes.

3.3. Values of Eight Information Units

The program shown in Listing 2 builds on the one of Listing 1. When started, it will ask the user to input the cardinality of an information unit. The cardinality of a bit is 2, because a single bit can assume the values 0 or 1. For a base of the genetic code the cardinality is 4, because there are four bases in the genetic code. Cardinalities smaller 2 are invalid. But the program does not take measures against invalid input.

```
1 // ValuesOfEightInformationUnits by Ulrich Eisenecker, July 15, 2020
2
3 #include <iostream>
4 using namespace std;
```

```

5
6 int main()
7 {
8     int cardinality;
9     cin >> cardinality;
10    cout << cardinality * cardinality *
11           cardinality * cardinality *
12           cardinality * cardinality *
13           cardinality * cardinality
14         << endl;
15    return 0;
16 }

```

Listing 2: `ValuesOfEightInformationUnits.cpp`

3.3.1. Variables

To allow the input, a new concept is introduced, namely a *variable*. In C++, a variable is a typed memory location that usually has a name. The line `int cardinality;` in Listing 2 defines the variable named `cardinality` being of the type `int`. As mentioned before, `int` is a predefined built-in fundamental type for representing signed integers with a limited range. The memory required by `cardinality` is automatically allocated. The name of a variable must follow the *naming scheme for identifiers* in C++. That is, it must start with an alphabet character or underscore. Following characters can be alphanumeric characters and underscore. However, user-defined identifiers should not start with an underscore, because these names are reserved by convention.

3.3.2. Input

The statement `cin >> cardinality;` performs the input. `cin` is an object, just as `cout`. It represents the *standard input*. The standard input is a stream of characters which is usually tied to a console window. `cin` is the left operand of the `>>` operator. The right operand of `>>` is the variable `cardinality`. The `>>` operator extracts as many characters of the input stream as required and as possible to build a value of the variable's type. Afterwards, the `>>` operator converts the extracted characters into a value of the required type and assigns it to its right operand, that is, the variable. Executing this line has the effect that the program waits for a valid input, e.g., 2. Usually, an input is terminated by pressing *Enter* or *Return* on a keyboard. Entering syntactically wrong input, e.g., alpha-characters, causes an error. How to address this issue will not be explained here. How to cope with semantically inappropriate input, e.g., 0 or 1 in this case, will be explained later.

As result of its execution the `>>` operator returns the left operand, that is, `cin`. Principally, this way further inputs can be realized, for example `cin >> variable1 >> variable2;`

3.3.3. Ignoring Return Values

An interesting question is, what happens, if the result which an operator or a function returns, is not used? The answer is: it is simply ignored. This is nice, if the result is not needed, but it may be harmful, if it should be used, but it is not. Fortunately, there is a possibility to control this more precisely. If a function is declared with the *attribute* `[[nodiscard]]` or `[[nodiscard("reason")]]` the compiler issues a warning if the result of the function invocation is ignored.

3.3.4. Line vs. Statement vs. Expression

A *line* is of minor syntactical importance. The compiler reports errors with respect to the line number, where it detects the error. A line is also important for the preprocessor. For example, the comment starting with `//` extends until the end of the line. Lines are important for humans, because they visually structure the source program.

A *statement* performs something. It can declare a variable, it can import the identifiers of a namespace, it can execute computations using operators or function calls. A statement may be *elementary*, such as `using namespace std;`, which is also called a *simple statement*. A *compound statement* is a block enclosed in curly braces, which can contain any number of statements. An *expression statement* executes at least one function or operator, but ignores the result. `cin >> cardinality;` is an example for an expression statement. Evaluating the expression yields the result `cin`, which is ignored by terminating the statement with a semicolon. Eventually, there is the `null` statement, also known as *empty statement*. It consists simply of a semicolon. The compiler processes it, but produces no code for it.

An *expression* executes a function or an operator or a combination of them. As such, a statement may contain an expression.

A statement or an expression may span over more than one line. In the program of Listing 2 the expression statement producing the output spans over five lines. It contains arithmetic sub expressions as well as sub expressions for producing the output.

In general, there is no relation between lines and statements or expressions.

Interestingly, with the exception of lines with preprocessor statements, a C++ program can be written in a single line. Of course, this makes it difficult to read and understand the program. Hence, one should refrain from writing such source programs.

3.3.5. Sequence

This program shown in Listing 2 is executed statement by statement in a strictly linear order, which also applies to the program of Listing 1. Statements that are executed one after the other form a so-called *sequence*. A sequence is one of the mandatory features of the *imperative programming paradigm*. Executing a sequence results in one linear execution path. Since the only function of this program is `main()` the whole program maps to a linear execution path.

3.3.6. Wrap-Up

The program in Listing 2 introduces two important features of a programming language, namely *variable* and *identifier*. Obtaining some input is not a new language feature. It is another instance of applying an operator as seen before, e.g., using the `<<` operator.

A variable increases the potential uses of the program. Instead of always computing the same value, it now computes values depending on some runtime input. The name of the variable was chosen to convey some of the meaning in the context of the given problem. Thus, *cardinality* is much more meaningful for a human than any literal value, e.g., 2. But the meaning of the number of occurrences of the operator for multiplications is only part of the filename of the source program. How to improve this, will be addressed later. Another issue is, that the program accepts values as input for which the subsequent computation is not meaningful. Of course, it could be argued, that with cardinality 0 only 0 values can be represented or that with cardinality 1 only 1 value can be represented. For a human user it would be more helpful to be informed, that these values are not useful. How to do this, will be addressed in the next section.

A sequence, which results in a linear execution path, was explicitly mentioned for the first time.

3.4. Values of Eight Validated Information Units

The program in Listing 3 is a revision of the program shown in Listing 2. It checks for invalid values for cardinality and takes measures if necessary. Furthermore, it is more verbose with respect to informing the user.

```
1 // ValuesOfEightValidatedInformationUnits by Ulrich Eisenecker, July 15, 2020
2
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     int cardinality;
```

```

9   cout << "Cardinality of information unit (must be >= 2): "
10      << flush;
11   cin >> cardinality;
12   if (cardinality < 2)
13   {
14       cerr << "Cardinality of information unit must be >= 2!\n"
15            << "Program terminated."
16            << endl;
17       return 1;
18   }
19   cout << cardinality * cardinality *
20         cardinality * cardinality *
21         cardinality * cardinality *
22         cardinality * cardinality
23         << " different values can be represented."
24         << endl;
25   return 0;
26 }

```

Listing 3: `ValuesOfEightValidatedInformationUnits.cpp`

3.4.1. flush

In lines 9 and 10 of Listing 3 the user is informed about the expected input. First, a string is sent to `cout` which informs the user about the expected input. Next, `flush` is sent to `cout`. As `endl` it is a manipulator. Sending `flush` to an output stream empties the buffer associated with the output stream, but it does not start a new line.

Sending `flush` to `cout` is not strictly necessary here, since the following input operation to `cin` automatically flushes the buffer associated with `cout`. This does not necessarily apply to other streams as well, and has no negative consequences other than a minimal loss of performance. Therefore, it is handled this way throughout this text.

3.4.2. Escape Sequences

The first string sent to `cerr` includes `'\n'` at its end. This sequence of two characters represents a *special character*. Such special characters have either special effects or special meaning. Here, `'\n'` represents the control character *linefeed*. Inserting it into a text starts a new line. Other examples are `'\t'`, which inserts a horizontal tabulator.

Special characters are represented by so-called *escape sequences*. An escape sequence always starts with `\` followed either by a normal character or by a sequence of characters representing a numeral value.

In `C++`, strings are always included in a pair of double quotes, `""`. Single characters are included in a pair of single quotes, `' '`. This also applies, if an escape sequence is used to represent a character, for example, `'\n'`.

To represent a double quote in a string, one also must use an escape sequence, that is `'\"'`. To represent a single quote as a character or as part of a string one must use the escape sequence `'\''`. The character `'\'` itself is to be represented as `'\\'`. For example, to output the string `\My name is "Jordan", not 'Morgan'.` (including the backslashes) the following statement is required:

```
cout << "\\My name is \"Jordan\", not \'Morgan\'.\\" << endl;
```

3.4.3. `'\n'` vs. `endl`

There are several ways to insert new lines into an output stream. The first has already been introduced, namely sending the manipulator `endl` to the output stream, for example `cout << "Yours sincerely," << endl << "Taylor" << endl;`. As mentioned earlier, `endl` inserts a newline character into the output stream and forces the internal buffer associated with the output stream to actually be written. This causes a slight overhead that can become noticeable when a plethora of output operations is performed with `endl`. The second option is to insert the newline character directly into the string to be output later, for example `cout << "Yours sincerely,\nTaylor" << endl;`. This avoids the overhead of flushing the output buffer more often than necessary. The resulting effect in the output stream is exactly the same. For this reason, several sources, e.g. (Breymann, 2023), p. 108, and (*C++ Core Guidelines*, n.d.), recommend avoiding the superfluous use of `endl` in general and using `'\n'` instead. In this text, this recommendation is followed only if the newline character is conceptually really a part of the respective string. That is, sending `endl` to the output stream would split a string into two parts that would otherwise naturally be a single string. In all other cases in this text, `endl` is sent to the output stream, since the corresponding strings are considered to be standalone strings. Regardless, no example programs occur in this text where outputting `endl` on an output stream would result in a relevant performance penalty compared to using `'\n'` as part of a string.

3.4.4. `if` Statement

Another novelty is the *if statement*. It begins in line 12 of Listing 3 and spans up to and including line 18. An if statement consists of a *condition* and a sub-statement. The condition is any expression which evaluates to a Boolean value or whose value can be automatically converted to a Boolean value. In C++, a Boolean value is represented by the built-in fundamental type `bool`. A Boolean value is either `true` or `false`. It may be automatically converted to the `int` values 1 or 0 if required. If necessary, any integer value other than 0 is automatically converted to `true`, while 0 is converted to `false`.

The sub-statement may be a single statement or a compound statement, i.e. it is a block enclosed in curly braces. It is only executed if the condition evaluates to true.

In the present case it is a compound statement. A compound statement has not to be ended with a semicolon. The compound statement outputs an error message and exits the function `main()`, returning 1. This value greater than 0 indicates that some error has occurred.

The `if` statement used in the program is one of two variants. The second variant is the *if-else statement* which requires another sub-statement after the keyword `else`. This sub-statement is executed, if the condition evaluates to false. Table 7 shows the schemas of the two `if` statements.

<i>if statement</i>	<i>if-else statement</i>
<pre>if (condition) single statement;</pre>	<pre>if (condition) single statement; else single statement;</pre>
<pre>if (condition) { statement; }</pre>	<pre>if (condition) { statement; } else { statement; }</pre>

Table 7: Schemes of `if` statements

Some programmers recommend to use `if` and `if-else` always in combination with compound statements as sub-statements. This helps to avoid programming errors due to inappropriate semicolons. Additionally, the compound-statement can be more easily extended.

The statement after `if` is also called *if branch*, the statement after `else` is called *else branch*.

This program of Listing 3 has two possible execution paths. The first is when the `if` condition evaluates to true, the second is when it evaluates to false. If the condition is true, the `if` branch is executed, which terminates the program with `return 1;`. Otherwise, the program continues with the statement immediately after the `if` statement.

3.4.5. Flowchart

The flowchart in Figure 9 gives a simplified graphical representation of the program. Symbols and syntax of flowcharts are explained in, e.g., (“Flowchart,” 2021).

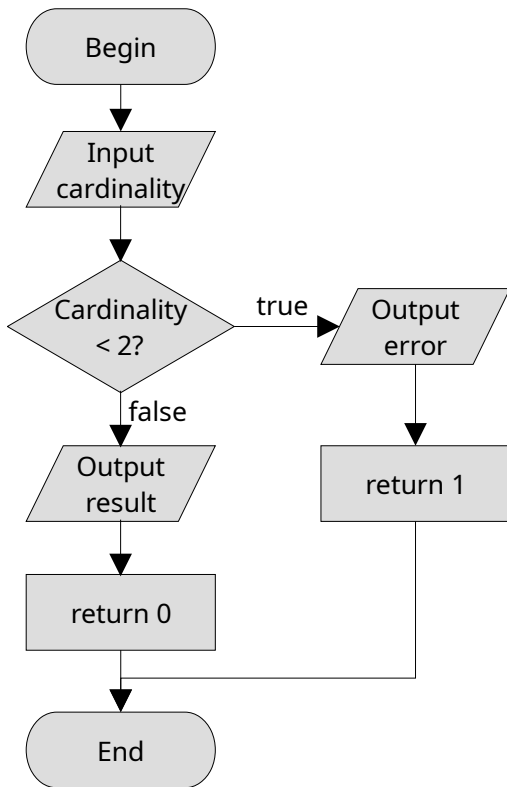


Figure 9: Flowchart of the program shown in Listing 3

The diagram of Figure 9 omits some details, for example, outputting a string asking for input or computing the result. To emphasize that it returns different values as result of function `main()`, two separate process symbols have been used.

Furthermore, the flow lines leaving the decision symbol are labelled *true* and *false*. In fact, the if branch of the program terminates the program when the condition evaluates to *true*. There is no else branch in the case the condition evaluates to *false*.

Of course, more endeavor could have been spent to make the program and the flowchart more consistent. In this case, the question is which one should be adapted, the flowchart or the program. Eventually, it often turns out that diagram and program cannot be made perfectly congruent without tweaking the structure or even semantics of one or the other. Thus, it is advisable to treat a diagram as a means to provide a coarse sketch of the program design to understand it or explain it to others.

3.4.6. Relational Operators

For all built-in numerical types, C++ defines *relational operators*, namely `<` (*smaller than*), `<=` (*smaller than or equal to*), `=` (*equal to*), `!=` (*not-equal to*), `>=` (*larger than or equal to*), and `>` (*larger than*). Each relational operator compares two numerical values. It returns either `true` or `false` as result. Furthermore, relational operators may be also defined for non-numerical types. The relational operators are complemented by the *three-way comparison operator*, `<=>`, also known as *spaceship operator* (because of its visual appearance). It has been introduced since C++20. To use it, the header `<compare>` must be included. It compares two objects lexicographically and returns a common category type as result. When available, it makes the definition of the other relational operators `<`, `<=`, `>=`, and `>` obsolete.

3.4.7. Standard Error Streams

`cerr` is – like `cin` and `cout` – an object representing a standard stream. `cerr` is the *standard error stream*, which is also connected to the console, but not buffered. Any output sent to it is immediately written to the stream. This is essential if an error crashes the program. Only then the user has a chance to see the error message in the console. An alternative to `cerr` is `clog`. `clog` is much the same as `cerr`, but it is buffered. Hence, error messages sent to `clog` may be lost when the program crashes.

3.4.8. Wrap-Up

The most important novelty is the `if` statement. In the concrete case it allows to detect syntactically correct, but semantically invalid input and to react accordingly. In general, it allows to write many more flexible programs. This will be pursued subsequently.

Moreover, the program in Listing 3 has become more verbose again. The user is informed about the expected input, invalid input, and the meaning of the computed value (line 23).

3.5. Values of Information Units

Listing 4 shows an even more flexible version of the preceding program. It allows to specify any number of bits and computes the number of values, which can be represented with it.

```
1 // ValuesOfInformationUnits by Ulrich Eisenecker, July 23, 2020
2
3 #include <iostream>
```

```

4 using namespace std;
5
6 int main()
7 {
8     int cardinality;
9     cout << "Cardinality of information unit (must be >= 2): "
10    << flush;
11    cin >> cardinality;
12    if (cardinality < 2)
13    {
14        cerr << "Cardinality of information unit must be >= 2!\n"
15        << "Program terminated."
16        << endl;
17        return 1;
18    }
19
20    int number;
21    cout << "Number of information units (must be >= 1): "
22    << flush;
23    cin >> number;
24    if (number < 1)
25    {
26        cerr << "Number of information units must be >= 1!\n"
27        << "Program terminated."
28        << endl;
29        return 2;
30    }
31
32    int values = 1;
33    while (number > 0)
34    {
35        values = values * cardinality;
36        number = number - 1;
37    }
38
39    cout << values
40    << " different values can be represented."
41    << endl;
42    return 0;
43 }

```

Listing 4: *ValuesOfInformationUnits.cpp*

3.5.1. At First Glance

The program has grown considerably. It is difficult to view and understand it in its entirety. Therefore, empty lines have been inserted to structure the source program into parts that are semantically relevant to the human programmer.

Probably the size of the program is at the borderline of the average cognitive capacity of human programmers. Most of the programming concepts introduced afterwards have the purpose to structure a complex program into smaller chunks. It is much easier for a programmer to read, understand, analyze, and maintain a small chunk of code in relative isolation.

The code sections asking for cardinality and number of information units share many commonalities. They differ only with respect to the name of the variable, the if condition, and the return value.

Computing the number of values introduces some new features. First, the explicit *initialization* of the value of a variable, the *while loop*, and the *assignment operator*, =.

3.5.2. Initialization of Variables

The two preceding programs define the variable cardinality. The statement `int cardinality;` declares the variable cardinality of type `int` and automatically allocates memory for it. In the following, the term *object* is used to refer to variables of any type without regard to where in memory they are located. This usage of the term *object* is not exactly the same as object in *object-oriented programming*, but there is a considerable overlap in meaning. ***Declaring an object and allocating its memory, means to define this object. Despite the fact that the object has just been defined, it has no defined value afterwards. A variable should not be read before a value is assigned to it.*** A value can be assigned to a variable, for example, by extracting a value from the standard input and assigning it to the variable. A definition without any arguments is conceptually equivalent to executing a *standard constructor*, also called *default constructor*.

Furthermore, it is possible to *initialize* an object when it is being defined, a so-called *copy initialization*. This can be done using the syntax `int values = 1;` or – fully equivalent – `int values(1);`. After a copy initialization the object has the provided value, here 1. A copy initialization with a value of the same type as the constructed object means conceptually to execute a *copy constructor*.

Another possibility for initialization is to use an *initializer list*, here `int values { 1 };`. In the context of an object definition an initializer list encloses none, one, or more values used for initialization in curly braces. In the given example, the statement `int values { 1 };` is equivalent to execute the copy constructor.

The empty initializer list, `{ }`, is the so-called *zero initializer*. It initializes integral variables to 0, floating point variables to 0.0, and other types by invoking their standard constructor.

It is highly recommended to use initializer lists. There is also strong advice to avoid objects which are not initialized. Defining objects with the zero initializer is a viable measure to achieve this. For example, in the program of Listing 4, the statement `int cardinality;` would have to be rewritten as `int cardinality { };` to follow this recommendation.

Each constructor that can be called automatically if an object of a specific type is expected, is a so-called *converting constructor* (*Converting Constructor - Cppreference.Com*, n.d.).

It should be noted that whenever a value is expected, any expression evaluating to a value of the desired type can be used. The statement `int values { 2 * 5 - 9 };` gives the same result as `int values { 1 };`.

Uninitialized objects can cause errors which are difficult to detect. For example, `int i;` creates a valid integer object, but without a defined value. If it is used in a computation without prior assignment, such as `cout << (i * 10) << endl;`, the result is unpredictable and may be different each time the computation is performed. Hence, it is recommended to always initialize an object when defining it. To achieve this, in Listing 4 line 8 would have to be rewritten as, for example, `int cardinality { 0 };` or – even shorter – `int cardinality { };`. Of course, this is only useful if the initialization value allows a valid execution of the subsequent program parts, or if it is checked whether the object is initialized with a certain value.

3.5.3. while Loop

A *while loop* starts with the `while` keyword, followed by a condition enclosed in parentheses and a statement, which can be a compound statement. The condition is any expression that evaluates to a Boolean value. When the condition is `true`, the subsequent statement is executed. Otherwise the program continues with the statement that follows the while loop. Since the while loop checks the condition first, the statement may not be executed at all.

To prevent a while loop from executing forever, either in its condition or in its statement a part of its condition must be altered such that the condition eventually evaluates to `false`. Here, the condition of the while loop is `number > 0`. As part of its compound statement, `number = number - 1;` is executed. This statement has the effect that in each iteration of the while loop `result` is decremented by one. Thus, after a certain number of repetitions `result` is guaranteed to become 0. In this case the condition evaluates to `false` and the while loop is no longer executed.

There are more looping constructs which will be discussed later.

3.5.4. Assignment Operator

The statement `number = number - 1;` makes use of the assignment operator, `=`. It has to be emphasized, that this is not an initialization of the variable `number`, because it is not declared in this context. Furthermore, the `=` operator should not be confused with the relational operator, `==`, testing for equality. Indeed, the statement should be read as *compute the actual value of number minus 1 and assign the result to number*.

Assigning a value to a variable is the usual way to change the state of a program in the imperative programming paradigm. Usually, an imperative program starts in an initial state. Then it executes commands performing computations and manipulating variables until a desired goal state is reached. Afterwards the program stops. Of course, there are programs which run forever until interrupted, for example, a program measuring temperature and switching on a heater if the measured temperature is below a threshold or switching it off when temperature is above.

3.5.5. Control Structures

All programming languages belonging to the imperative programming paradigm have three mandatory so-called *control structures*:

1. *Sequence* – statements of a sequence are executed in linear order.
2. *Branching construct* – depending on a condition some code is executed or not.
3. *Looping construct* – depending on a condition a statement is executed repeatedly.

Any imperative programming language that has at least one variant of each of these control structures is considered Turing-complete in a practical sense. That is, the compound statement, the one-way if statement, and the while loop can be used to implement any algorithm that can be implemented by any other Turing-complete programming language.

In view of this, the introduction to programming could be finished now – if it were only a matter of developing programs for computers!

However, this is not the case. A computer executing a program has no idea of its meaning. But human programmers who develop and maintain a program essentially rely on the meaning conveyed by identifiers, selected constructs, and comments. Important properties of a program are thus:

- It is easy to read.
- It is easy to analyze.
- It is easy to adapt to changing requirements.
- It is easy to integrate new requirements.

These qualities depend largely on additional means and concepts that facilitate or ensure the writing of programs that exhibit these properties. In addition, powerful programming concepts allow us to implement abstraction layers that can perform domain-specific optimizations that would otherwise be impossible to achieve.

These are the reasons why most of this introduction to programming is still ahead.

3.5.6. Wrap-Up and Outlook

So far, all the constitutive features of the imperative programming paradigm have been introduced, namely variables, reading variables, assigning values to variables, calculating and the three mandatory control structures sequence, branching, and looping. Together, these components form a Turing-complete programming language. However, the program in Listing 4 suggests that developing programs with thousands of lines of code using these resources only is insufficient for human developers and does not support the above qualities.

When programs are no longer developed by humans, these qualities become less important, for example, when an artificial neural network optimizes a source program with respect to some functional criteria. The resulting source program may be incomprehensible to human experts, but outperform any other human-developed program. (Karpathy, 2017) outlines a corresponding approach to software development which he calls *Software 2.0*. However, with the advent of large generative language models such as *ChatGPT* (*ChatGPT*, n.d.), this has once again changed dramatically. They are able to generate well-formatted, understandable programs based on natural language specifications. Usually, these programs can also be compiled without errors. Nevertheless, they need to be thoroughly checked and tested, as they sometimes contain subtle errors that are difficult to detect. It is a common practice to use them as coding assistants.

The next section introduces a fourth but optional construct of imperative programming: the *procedure* and *procedure call*. They are called *function* and *function call* in C++.

3.6. Functions

The visual structure of the program shown in Listing 4 reflects its semantic structure:

1. Input cardinality
2. Input number of information units
3. Calculate number of representable values
4. Output representable values

To facilitate understanding, this structure could be recorded in the form of comments before the corresponding lines of code. An example is shown in Listing 5.

```
1 // Input cardinality
2 int cardinality;
3 cout << "Cardinality of information unit (must be >= 2): "
4   << flush;
5 cin >> cardinality;
6 if (cardinality < 2)
7 {
```

```

8     cerr << "Cardinality of information unit must be >= 2!\n"
9         << "Program terminated."
10        << endl;
11    return 1;
12 }
13 // ...

```

Listing 5: *ValuesOfInformationUnits.cpp with comment (excerpt)*

Another possibility is to convert the corresponding code sections into functions. Listing 6 shows the result.

```

1 // ValuesOfInformationUnitsWithFunctions by Ulrich Eisenecker, August 8, 2024
2
3 #include <iostream>
4 using namespace std;
5
6 int inputCardinality()
7 {
8     int cardinality { };
9     do
10    {
11        cout << "Cardinality of information unit (must be >= 2): "
12            << flush;
13        cin >> cardinality;
14        if (cardinality < 2)
15        {
16            cout << "Illegal value - please, retry!"
17                << endl;
18        }
19    }
20    while (cardinality < 2);
21    return cardinality;
22 }
23
24 int inputNumberOfInformationUnits()
25 {
26     int number { };
27     do
28     {
29        cout << "Number of information units (must be >= 1): "
30            << flush;
31        cin >> number;
32        if (number < 1)
33        {
34            cout << "Illegal value - please, retry!"
35                << endl;
36        }
37    }
38    while (number < 1);
39    return number;
40 }
41
42 int calculateRepresentableValues(int cardinality,int number)
43 {
44     int values { 1 };
45     while (number > 0)
46     {
47        values = values * cardinality;
48        number = number - 1;
49    }
50    return values;
51 }
52
53 void outputRepresentableValues(int values)
54 {

```

```

55     cout << values
56         << " different values can be represented."
57         << endl;
58 }
59
60 int main()
61 {
62     int cardinality { inputCardinality() };
63     int number { inputNumberOfInformationUnits() };
64     int values { calculateRepresentableValues(cardinality,number) };
65     outputRepresentableValues(values);
66     return 0;
67 }

```

Listing 6: ValuesOfInformationUnitsWithFunctions.cpp

3.6.1. inputCardinality() Function

A function consists of a *header* and a *body*. Here the header is `int inputCardinality()`, the body is the remainder enclosed by a pair of curly braces, `{ ... }`, that is, a block. Together, the header and the body form the *definition of a function*.

The name of the function is `inputCardinality`. The preceding `int` specifies the type of the value the function returns. The following pair of parentheses contains the parameters that are passed to the function. Since no parameters are specified here, the function does not take any parameters. The name of the function has a meaning for the programmer. It says that an input is expected, and it specifies what input is expected. Its spelling corresponds to the so-called *camel-case notation*. That is, it starts with a lowercase letter and the first letter of following word is capitalized. This is purely a convention. An alternative convention is to use only lowercase letters and separate the words with an underscore, for example `input_cardinality`. ***It is less significant which convention to use. But it is essential to use it consistently.***

It is another convention in written text to refer to a function name with an appended pair of parentheses, e.g., `inputCardinality()`. This clarifies that this identifier denotes a function and not something else, e.g., a variable or a type.

The body of the function is always a compound statement. The first statement inside the body is the definition of the variable `cardinality`, which is of type `int`. The variable exists only within the function during its execution. When the function is exited, the variable is destroyed. Hence, it is a *local variable*. The name of a local variable hides a variable of the same name which is declared in a superordinated scope. For example, in Listing 7, the variable `i` in the inner block hides the variable `i` of the outer block. Therefore, this code snippet outputs 99 when executed.

```

1 {
2     int i = 1;
3     {
4         int i = 99;

```

```

5     cout << i << endl;
6   }
7 }

```

Listing 7: *Blocks and hiding identifiers with the same name (excerpt)*

Next, a new loop statement is introduced, the *do-while loop*, or shorter, the *do loop*. It consists of the keyword `do`, followed by a statement, and the keyword `while`, which specifies a condition expression in parentheses. As long as the condition evaluates to `true`, the loop repeats. If the condition evaluates to `false`, the loop is exited. A `do` loop is executed at least once.

The actual `do` loop is executed as long as the value of the `cardinality` variable is less than 2. The loop statement is a compound statement. First, it informs the user of the value to enter. Then it reads the user input and stores it in the `cardinality` variable. Then it uses an `if` statement to check whether the value of `cardinality` is less than 2, and if it is, it prints an error message. Accordingly, the following condition of the `do` loop is evaluated as `false`, and the user is prompted for input again.

There are several points to mention:

- The `if` statement uses a compound statement. This is in accordance with the previously given advice to generally use compound statements in this case. Syntactically, a simple statement would have sufficed here.
- The error message is sent to `cout`, not to `cerr`. It informs the user about the incorrect input. This makes sense because it is no longer an error message explaining why the program was aborted.
- In the case of an error, neither the function is exited nor the program is terminated. The user is repeatedly asked for an appropriate input.

3.6.2. `inputNumberOfInformationUnits()` Function

This function has the same structure as `inputCardinality()`. Therefore, no detailed explanation is given.

3.6.3. `calculateRepresentableValues()` Function

The name of this function indicates that it calculates the number of different values that can be represented using number information units of `cardinality`. Therefore, `cardinality` and `number` are passed as *parameters*. Each parameter is specified by its type and name, under which it is provided in the function. A following parameter is to be separated by a comma from its preceding parameter.

Function parameters in `C++` are *position parameters*. This means that when a function is called, ***the order of the parameters must exactly match the order of***

their declaration in the function header. It is the responsibility of the programmer to pass the parameters in the correct order. Otherwise, the function may be executed with incorrect values.

Sometimes a parameter declared in a function header is called a *symbolic parameter*, while the actual variable or expression passed when a function is called the *actual parameter*. A function that is called is also called as *callee*, while the piece of code that calls the function is called a *caller*.

The name of a parameter is valid as long as the function is executed. Like a local variable, it may hide an identical name in a superordinated scope.

In C++, parameters are passed by value by default. This means that the value of the variable (or the expression) passed as a parameter is copied and the copy is made available within the function under the name with which it is declared in the function header. When a function is exited, a value parameter is destroyed and no longer available. A value parameter can therefore also be considered a local variable.

The first statement of the function `calculateRepresentableValues()` defines the local variable `values` and initializes it with the value 1. This initialization is necessary because in the following while loop the product of the current values of `values` and `cardinality` is assigned as new value to `values`. The second statement in the compound statement of the while loop decreases the value of `number` by 1. In this way, the condition (`number > 0`) of the while loop is finally evaluated as `false` and the while loop ends. Changing the value of `number` only affects the local copy of the current parameter with which the function was called. The actual parameter (outside the function) remains unchanged. This is a consequence of passing the parameter `number` by value.

The last statement of the function returns the value of `values` as the result. Conceptually, the result of a function is returned by value by default. That is, the value of a local variable or expression is copied and the copy is returned as the result.

3.6.4. `outputRepresentableValues()` Function

This function specifies its return type with the keyword `void`. `void` is a so-called *incomplete type*, which means that “*nothing special can be said about this type*”. ***A function with return type `void` returns nothing.***

A function in C++ with the return type `void` corresponds to a procedure in the imperative programming paradigm. A function returning a type other than `void` corresponds to a function in imperative programming. In the imperative programming paradigm, operation is the parent concept of function and

procedure. Unfortunately, almost every programming language has its own specific vocabulary for its concepts. In general, the same term may denote different concepts in different programming languages, and the same concept may also differ between different programming languages.

The `outputRepresentableValues()` function has one parameter, `values`, which is passed by value. The value of this parameter is sent to `cout`.

Since the function does not return a value, it simply terminates after its last statement is executed. In fact, it would be legal to explicitly terminate the function by adding `return;` as the last statement. Although this makes it clearer what is happening, most programmers omit a superfluous `return;`.

3.6.5. `main()` Function – More Information

The first statement of `main()` defines the variable `cardinality` of type `int` and initializes it with the result of the call to `inputCardinality()`. The name `cardinality` defined in `main()` does not conflict with the name `cardinality` defined in `inputCardinality()`, because the names are defined in different scopes. A *scope* is a part of a program where a certain definition is available. For example, the block of a function definition is a scope. When a block is defined within another block, it also defines its own nested scope. Therefore, `cardinality` defined in `main()` is different from `cardinality` defined in `inputCardinality()`.

The second statement defines `number` and initializes it with the result of the call to `inputNumberOfInformationUnits()`.

The third statement defines `values` and initializes it with the result of the call to `computeRepresentableValues()`. The variables `cardinality` and `number` are passed as parameters to `computeRepresentableValues()`. Two important points should be mentioned:

- The names `cardinality` and `number` are defined and valid in the scope of `main()`. These names are no longer relevant when used as actual parameters for the call to `computeRepresentableValues()`. In `computeRepresentableValues()` these parameters are referred to by the names of the symbolic parameters which have been defined in the header of this function.
- On the caller's site it is not apparent how the parameter is passed, e.g. by value. This information is only available at the callee's site, namely in the function header, where the parameters and the way, how they are passed are defined.

The fourth statement calls `outputRepresentableValues()` and passes `values` as parameter to output the calculated result.

The last statement is `return 0;`. It terminates `main()` and returns 0 as the result, indicating that `main()` terminated without error. In fact, `return 0;` can be omitted. `main()` will always return 0 if no return statement is specified, which is only allowed for the `main()` function. This is only similar but not equal to the case that a function that returns `void` does need not specify a return statement at all.

Overall, `main()` has a clear structure. First, all the values needed for the subsequent calculations are entered, then the desired calculations are performed, and finally the results are printed. This structure, *input – calculate – output*, is shared by many programs running in a console window.

3.7. Wrap-Up

Functions were introduced as a concept to structure a program into smaller parts that can be referred to by a name that makes sense to the programmer. As such, they are indispensable for writing programs that are easier for the human programmer to understand, thus facilitating the maintenance and reuse of program parts.

They can also provide parameters that extend their possible uses. A function can be called with a large number of specific parameter values and, of course, with a large number of combinations of different parameter values.

Call by value was introduced as a standard mechanism of passing parameters to functions.

The *do loop* was introduced as another loop construct. It is executed at least once.

To complement this section, the activities *testing* and *debugging* are presented next.

3.8. Testing

Testing is a crucial activity in software development. A naive approach would be to check whether a program or a part of it behaves as expected. In the case of the program shown in Listing 6, this could mean that after entering 2 as *cardinality* and 2 as *number of information units*, it outputs 4 as *different values*.

But how meaningful is the result of this test? If the program outputs a result that differs from 4, it must be faulty, assuming that the input was correct and the calculation was correct. But how sure is it that it is error-free if it outputs the expected result?

If the type `int` were represented with 16 bits, there are $2^{16} = 65,536$ different values for each, *cardinality* and *number*. Would it be sufficient to test all values for one

variable and keep the other constant? The answer is no, because there are $2^{16} \cdot 2^{16}$ different pairs of values that need to be tested for an *exhaustive test*. For this reason, exhaustive tests are almost impossible in practice. Therefore, there are different testing strategies and methods for different purposes. In the following, unit tests are presented in more detail. Finally, *regression testing* and *automated unit testing* will be discussed.

3.8.1. Unit Tests

The terms *unit* and *module* are often used as synonyms. When it comes to testing, the term *unit* is the dominant one.

Ideally, a module can be used and understood in relative isolation. A module's interface should have only essential dependencies on other modules, i.e., it should have minimal coupling. The implementation of a module should serve exactly one clear purpose, i.e., it should have maximal cohesion. The implementation may consist of nested modules or use other modules.

A function is the most essential unit of a program. Its header corresponds to the *interface* and its body to the *implementation* of a module.

A *unit test* checks a function in isolation. Often, a minimal environment for evaluating a function must be prepared, such as variables, or more complex data structures. Then the function is executed with certain parameters. If *expected* and *actual result* differ, both the function and its test are examined to find the cause of the discrepancy. Especially if the test was written recently and executed for the first time, the test itself may contain errors.

A special situation arises when a function creates a so-called *side effect*, e.g., by assigning a value to a global variable or performing outputs and inputs to and from external files. A *global variable* is a variable defined before a function in the global scope. ***Global variables should be used restrictively and with the utmost care.*** Examples for global variables are `cin`, `cerr` and `cout`. Reading values from standard input and writing values to standard output have already been introduced. In these cases, implementing a unit test can become difficult and requires special measures.

In the program shown in Listing 6, `inputCardinality()` and `inputNumberOfInformationUnits()` read only one value from standard input and return it. `outputRepresentableValues()` outputs only its function parameter on standard output and returns `void`, that is, nothing. Therefore, no unit test for these functions is demonstrated here.

`main()` is also a function that could be subjected to a unit test. However, it uses three functions with side effects. Therefore, no unit test for `main()` is presented here. Furthermore, unit tests for `main()` are not common. `main()` represents the entry

and exit points of a program. As such, it is rather the target of *integration tests*, which will not be explained here.

`calculateRepresentableValues()` takes two parameters and computes the number of different values that can be represented based on the parameters. Therefore, it is a good candidate for unit testing.

Table 8 shows the value pairs for the test of `calculateRepresentableValues()` and the expected results.

Description	Cardinality	Number	Expected Result
Two smallest values	2	1	2
One smallest, one larger value	2	4	16
One larger value, one smallest value	3	1	3
Two larger values	3	4	81

Table 8: Expected results for testing `calculateRepresentableValues()`

Table 8 contains only values for *Cardinality* and *Number* that are equal to or greater than the respective lower bounds, i.e., 2 or greater for *Cardinality*, and 1 or greater for *Number*. Legal values that give results outside the range are not tested.

The strategy for selecting pairs of values to test here is to cover all combinations of legal smallest and a legal larger value.

The program in Listing 8 shows a straightforward implementation of the four test cases using only that programming constructs already presented.

```
1 // UnitTestOfCalculateRepresentableValues by Ulrich Eisenecker, August 8, 2024
2
3 #include <iostream>
4 using namespace std;
5
6 int calculateRepresentableValues(int cardinality,int number)
7 {
8     int values { 1 };
9     while (number > 0)
10    {
11        values = values * cardinality;
12        number = number - 1;
13    }
14    return values;
15 }
16
17 int main()
18 {
19     int passed { 0 };
20     int failed { 0 };
21
22     cout << "Unit testing of calculateRepresentableValues()\n" << endl;
23
24     cout << "Test case \"Two smallest values\" ... "
25          << flush;
26     if (calculateRepresentableValues(2,1) == 2)
```

```

27     {
28         passed = passed + 1;
29         cout << "passed." << endl;
30     }
31     else
32     {
33         failed = failed + 1;
34         cout << "failed." << endl;
35     }
36
37     cout << "Test case \"One smallest, one larger value\" ... "
38         << flush;
39     if (calculateRepresentableValues(2,4) == 16)
40     {
41         passed = passed + 1;
42         cout << "passed." << endl;
43     }
44     else
45     {
46         failed = failed + 1;
47         cout << "failed." << endl;
48     }
49
50     cout << "Test case \"One larger, one smallest value\" ... "
51         << flush;
52     if (calculateRepresentableValues(3,1) == 3)
53     {
54         passed = passed + 1;
55         cout << "passed." << endl;
56     }
57     else
58     {
59         failed = failed + 1;
60         cout << "failed." << endl;
61     }
62
63     cout << "Test case \"Two larger values\" ... "
64         << flush;
65     if (calculateRepresentableValues(3,4) == 81)
66     {
67         passed = passed + 1;
68         cout << "passed." << endl;
69     }
70     else
71     {
72         failed = failed + 1;
73         cout << "failed." << endl;
74     }
75
76     cout << "\nTotal test cases: " << (passed + failed) << endl
77         << "Passed test cases: " << passed << endl
78         << "Failed test cases: " << failed << endl;
79 }

```

Listing 8: UnitTestOfCalculateRepresentableValues.cpp

The `calculateRepresentableValues()` function, which is the subject of the test, had to be copied from the original source file. *Copy & paste* has a long tradition in software development, but it also has its drawbacks. If an error is found during testing, it must be fixed consistently in both the original and the copies. Later, it will be explained, how functions can be kept in separate files, compiled to separate so-called *object files*, and finally linked to executable applications. For now, copy & paste will have to suffice.

Lines 19 – 22 of Listing 8 are needed to record the number of passed and failed tests and to inform the user about the subject of the tests. After that, the four test cases are handled. No special code is required to prepare the tests of `calculateRepresentableValues()`.

All test cases have the same structure:

- The user is informed about which test case will be executed next.
- The condition of the if statement checks whether the result of the execution of the function with the parameters of the test case equals the expected result. For this purpose, the `==` operator is used, which checks for equality of its operands.
- The if branch increments `passed` and informs the user that the test case passed.
- The else branch increments `failed` and informs the user that the test case failed.

At the end, the program outputs how many test cases were executed in total, how many passed, and how many failed. Since `main()` does not contain an explicit return statement, it implicitly returns 0.

Unit tests are critical for all non-trivial units of a program, e.g., functions that perform calculations, complex input or output operations involving formatting and/or conversions. The goal is to ensure that all relevant units of a program are adequately tested before the program is integrated. After that, integration testing, performance testing, load testing, robustness testing, and acceptance testing become relevant. Only in the case of unit tests it is common for them to be written and executed by the programmers themselves. However, unit tests must be well planned and carefully implemented, which requires at least sound guidance or sufficient experience.

3.8.2. Regression Tests

As a rule, unit tests should be repeated frequently, e.g. after editing the source code or before a nightly build of the entire application. This testing strategy is called *regression testing*. Regression testing is an effective way to prevent errors that were previously fixed from reoccurring later.

3.8.3. Automated Unit Tests

The general and repetitive structure of unit tests and the need for frequent unit tests suggest automating unit tests. There are many tools for unit testing for

different programming languages. Writing *automated unit tests* using so-called *test frameworks* and executing them will be covered later.

3.9. Debugging

Testing is very useful, if not essential, for detecting bugs. However, sometimes it is necessary to understand exactly the cause of an error or how the error propagates through a program. There may also be cases where it is desirable to understand the behavior of a program at runtime compared to its source code.

The activity used to understand the circumstances and consequences of an error in a program or the exact execution of some parts of a program is called *debugging*. A software tool that supports debugging a program is called a *debugger*. Basically, debuggers are pure command line tools. Usually they are equipped with a graphical user interface that makes their use more convenient.

3.9.1. From Source Code to Executable Program

After the source code of a program is completed and stored on a storage medium, it is compiled. The compiler creates an executable program that can be run by entering its name in a console window or by selecting and launching it in a file manager.

The executable program is a long chain of bits. These bits represent either instructions or data. However, there are other steps and intermediates. It is important to know more about them for debugging.

When compiling a C++ program the following steps are performed:

- The preprocessor prepares the source code for the actual compilation.
- The compiler scans the source code for *tokens*, and parses it into an *abstract-syntax tree*, which can then be optimized. Finally, the *object code* is generated. Today, most compilers generate object code directly. If desired, a compiler can generate a so-called *assembler file*, which contains *assembler instructions* and data. The assembler file is processed by an *assembler* which generates the object code.
- In most cases, a program contains references to other object code or libraries. A *linker* takes the program's object code and links it to other code or libraries. There are two types of libraries that are used differently. The first type is a *static library* that is statically linked to the program code. The second type is a *dynamic library* that is dynamically linked when the referencing program is executed. Each program with a statically linked library has its own instance of the library. All programs that use a dynamic library usually share a single

instance of the library. A dynamic library must be installed separately and be accessible to programs that rely on it.

- The operating system *loader* loads a program to a specific memory location and can perform additional tasks. The loader is started implicitly when the program is executed in a console window or with a file manager.

Figure 10 is an example of a hypothetical abstract syntax tree for the statement $a = (b + c) * 4;$.

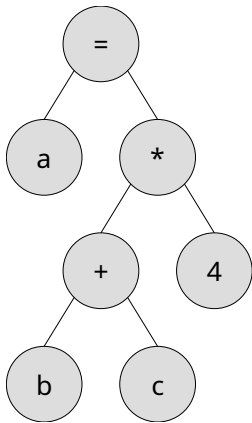


Figure 10: Abstract syntax tree for $a = (b + c) * 4;$

The assignment operator, =, has the lowest priority of all operators. It therefore forms the top node. It has two operands, namely the variable on the *left-hand side* (short: *lhs*), the target of the assignment, and the expression on the *right-hand side* (short: *rhs*). Before the assignment can be made, both operands must be evaluated. For the left operand this is easy because it is a variable. For the right operand it is more complicated, because it is a complex expression. The parentheses have a higher priority. Therefore, the * operator has the lowest priority and is at the top of the subtree. Its left operand is the + operator and its right operand is the literal 4. Finally, b and c become the leaves of +. The actual data structure of an abstract syntax tree, which is managed by the compiler, is much more complex. It also contains, for example, links to definitions, types, references, and uses.

Preprocessing and assembly can be viewed as further instances of mappings between problem and solution spaces. This is illustrated in Figure 11.

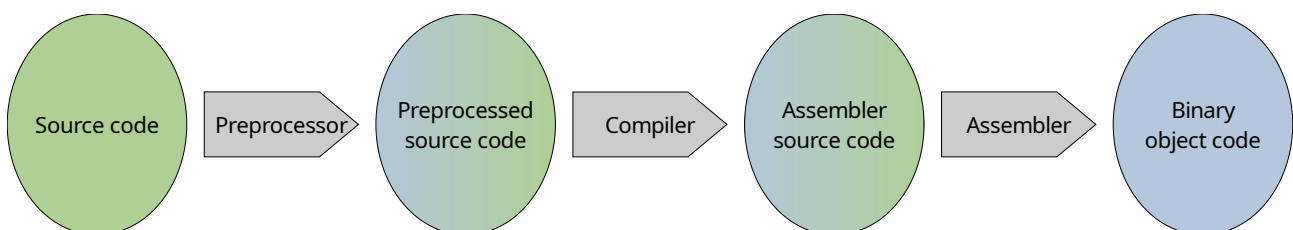


Figure 11: Chained mappings of problem and solution spaces

Basically, the source code is a result of mapping the problem space (application domain) to the solution space (solution domain). The source code is first considered as a specification in its own problem space. It contains parts that specify how it should be transformed by the preprocessor. The preprocessor takes this specification and creates a preprocessed source file that belongs to an intermediate solution space. At the same time, it is also a specification of an intermediate problem space, which serves as a specification that is processed by the compiler. The compiler generates assembly source code, which is the closest textual representation to binary object code. The assembly source code is in turn both a solution in an intermediate solution space and a specification in an intermediate problem space. Finally, the assembler performs the final conversion of the assembly source code into the binary object code.

In this way, an extremely complex mapping from the problem space to the solution space is decomposed into a sequence of less complex mappings. In addition, all mapping operations from source code to object code are fully automated. ***Automation of software development is essential for improving productivity and quality.***

This complexity is usually not visible to the programmer. Nevertheless, it is indispensable to know it in order to get at least a basic understanding of what goes on behind the scenes, for example:

- Building large software systems from individually compiled pieces of code.
- *Profiling* a program, i.e. examining its performance in case of problematic resource consumption and looking for suitable optimizations.
- Debugging a program to gain a deeper understanding of the program.
- Debugging a program to find the cause of an erroneous behavior.

3.9.2. Preprocessor

In principle, a preprocessor is not dependent on a specific programming language. However, the *C* and *C++* programming languages are closely tied to the *C* preprocessor, called *cpp*. In the future, the preprocessor for *C++* is to become superfluous.

A preprocessor directive begins with a hash character, *#*. The main categories of preprocessor directives are *source file inclusion*, *conditional inclusion*, and *macro definition*. The *C* preprocessor also performs *macro substitution*, that is, it replaces the use of a previously defined macro with the text resulting from the expansion of the macro. The preprocessor also removes any comment, i.e. *//* to the end of the line, or any text that is between */** and **/*, including delimiters.

Thus, if a source program contains a preprocessor directive, the compiler never sees the program as it was written by the programmer. This has a serious consequence: The compiler knows nothing about the original source program! It sees only the source program generated by the preprocessor. Anything that has been modified by the preprocessor cannot be debugged appropriately.

Normally a C++ compiler also provides the possibility to generate preprocessed output. For example, one can open a console window, change to the directory with the source programs presented so far, and run `g++ -E ValuesPerByte.cpp`. The `-E` option causes `g++` to perform only the preprocessing step. If no mistake was made, the result is a very long output. This is the source program that is actually processed by the compiler!

To preserve this output, `g++ -E ValuesPerByte.cpp > ValuesPerByte.ii` must be executed, or alternatively `g++ -E -o ValuesPerByte.ii ValuesPerByte.cpp`. In the console window, `>` redirects the standard output to the file named after it. In this case the file is named `ValuesPerByte.ii`. The compiler option `-o` followed by a filename specifies the name of the output file. `ValuesPerByte.ii` is significantly larger than 1 MB. Therefore opening this file with a text editor may take a while. It is only at the end of this file that most of the original program text appears. There is exactly one line missing, namely `#include <iostream>`. All the text before the original program is the result of executing this single preprocessor instruction!

3.9.3. Assembler

The first step of compilation is performed by the *lexer*, which is also called the *scanner*. It breaks down the source program into *tokens*. A *token* is an elementary part of a source program in terms of program syntax, for example a keyword, an identifier or an operator.

Then the parser tries to find tokens that belong together, e.g. a statement, a function definition, a variable definition and so on. The found constructs are inserted into the abstract syntax tree. The abstract syntax tree (see Figure 10) is a tree-like data structure that eventually represents the entire program. It is then unparsed directly into machine code or alternatively into assembler source code. The assembler source code contains data and so-called *mnemonics*, i.e. textual representations of processor instructions.

Normally a C++ compiler allows the creation of an assembler file. For example, in a console window one can change to the directory containing the source programs presented so far and execute `g++ -S -o ValuesPerByte.s ValuesPerByte.cpp`. The `-S` option causes only assembler language output and the `-o` option and the following filename specifies the output file. `ValuesPerByte.s` has more than 5,000 bytes, which

is larger than the corresponding source file, but much smaller than the file generated by the preprocessor. To check the file, it can be loaded into a text editor.

Of course the assembler source code can be processed further. By entering `g++ ValuesPerByte.s` in the console window the assembler and then the linker is executed. Afterwards an executable file named `a.out` is located in the active directory. The use of the suffix `.s` for the file name of the assembler source is essential in this case.

The assembly source code is the textual representation of a program that most closely resembles the executable binary program. If a debugger were to operate only on assembly code, it would be almost impossible for a human to cognitively trace the assembly instructions to the corresponding instructions in the source code. Therefore, when compiling a program, care must be taken to ensure that the required source code is also included. With `g++`, for example, the `-g` option serves exactly this purpose. If one executes the line `g++ -g ValuesPerByte.cpp` in a console window, the executable file `a.out` is created in the active directory. Now `a.out` contains debugging information.

Only the parts of a program that have been compiled specifically for debugging can be debugged at source code level. All other parts can only be debugged on assembler level. There is a close correspondence between assembler source code and binary code. Therefore binary code can be disassembled with a so called *disassembler*. Of course, if the assembly source code is written by programmers, they can use symbolic names that resemble macros. But all symbolic names in the assembler source code assigned by the programmer are lost when assembling. They cannot be recovered by the disassembler. Instead, they are replaced by automatically generated names.

3.9.4. Linker

A program requires a general amount of code to be executable at all. Also, programs usually refer to elements that are in other binaries. The linker combines the compiled part of the program that the programmer wrote with all the other required parts and creates an executable file. This process is called *linking*.

Internally, the source code is compiled into *binary object code*, which has a binary format. Then the various binary object code files, which together form the entire program, are linked into an *executable object code file*, or *executable*, which is also a binary format.

Normally a `C++` compiler provides the possibility to generate binary object code only. For example, one can open a console window, change to the directory containing the source programs presented so far, and run `g++ -c ValuesPerByte.cpp`.

The `-c` option suppresses the execution of the linker. So only a binary object code file is created. In this case it is called *ValuesPerByte.o*. It is slightly larger than 10,000 bytes. Despite its binary format it cannot be executed.

Nevertheless, it can be linked afterwards. Running `g++ ValuesPerByte.o` in a console window creates the executable binary code *a.out*. Running `g++ -o vpb ValuesPerByte.cpp` causes the compiler to create an executable file named *vpb*. As before, the `-o` option tells `g++` the name of the file to generate. On Windows, the executable should be named *vpb.exe*. On Windows, executable files have the extension *.exe* by default.

It is possible and often necessary to develop a program piecemeal. The result is several object files that must then be linked to form an executable file. For example, `g++ part1.o part2.o part3.o -o myprog` will link the object files *part1.o*, *part2.o*, and *part3.o* to an executable binary called *myprog*. Later it will be explained how to split a program into parts that can be compiled separately.

3.9.5. Example for Debugging

Now the necessary background for debugging has been presented. To demonstrate debugging, a program for calculating the checksum, also known as sum of digits, is used (Listing 9). It reads in a natural number, calculates the checksum, and outputs the result.

```
1 // Checksum by Ulrich Eisenecker, August 2, 2020
2
3 #include <iostream>
4 using namespace std;
5
6 unsigned long checksum(unsigned long number)
7 {
8     if (number % 10 == number)
9         return number;
10    else
11        return (number % 10) + checksum(number / 10);
12 }
13
14 int main()
15 {
16     cout << "Natural number: " << flush;
17     unsigned long number { };
18     cin >> number;
19     cout << "Checksum = " << checksum(number) << endl;
20     return 0;
21 }
```

Listing 9: *Checksum.cpp*

In this program some new features are introduced:

- `unsigned long` is an integer data type with a composite name. By using `unsigned` it represents *natural numbers*, i.e. non-negative integers, in a

limited range. The lower limit of this range is 0, the upper limit is calculated according to the formula $upper\ limit = 2^{(number\ of\ bytes\ (data\ type) \times 8)} - 1$. Because of long it must use at least as many bytes as int, yet it usually occupies more bytes. The C++ standard defines that int has a width of at least 16 bits, and long will have at least 32 bits. The two type names unsigned long and unsigned long int are completely equivalent. For the compiler it makes no difference which name is used. Some programmers prefer the longer name because it is more specific with respect to integral numbers, others prefer the shorter name because it is more concise but still unique.

- The modulo-operator, %, calculates the remainder of an integer division, e.g., the result of `28 % 5` is 3.
- The division-operator, /, when applied to integer values, performs integer division, i.e. the result is an integer value and all decimal places of the result are lost. For example, the result of `28 / 5` is 5, not 5.6. In C++ and also in many other programming languages, multiplication and division have a higher priority than addition and subtraction. This is also true for the modulo operator. Therefore, the parentheses around the modulo operation in `return (number % 10) + checksum(number / 10);` could have been omitted. Nevertheless, the additional parentheses emphasize the structure of the arithmetic expression, which makes it easier to understand the code.
- The `checksum()` function has a recursive implementation. The if statement checks, whether number modulo 10 is equal to number. If it is, the if branch returns number as the result. Otherwise, the else branch returns number modulo 10 plus the return value of the call to the `number()` function itself, where number is passed integrally divided by 10 as a parameter. A function that calls itself in its implementation is called a *recursive function*. A function calling indirectly itself is also a recursive function. More precisely, it is an *indirectly recursive function* (see Turing-Completeness).

There are many debuggers, most of which are integrated into a so-called *integrated development environment, IDE* for short. *gdb* is the stand-alone debugger of the *GNU project*. It is an open source, text-based debugger that runs in a console window. There are several graphical frontends for *gdb* that are more convenient to use. For the screenshots shown in Figures 12 – 14, the graphical frontend *nemiver* was used. In principle, any debugger can be used to reproduce the following example. For now, the focus is on understanding the concepts of debugging. It is advisable to reproduce the examples later.

The following example uses *g++* as compiler, *gdb* as debugger and *nemiver* as graphical frontend.

To prepare the program from Listing 9 for debugging, it must be compiled with the `-g` option, i.e. `g++ -g -o cs Checksum.cpp`. The `-o` option followed by the filename `cs` creates an executable program named `cs`.

Then *nemiver* cs is executed in the console window. Figure 12 shows the window that appears afterwards.

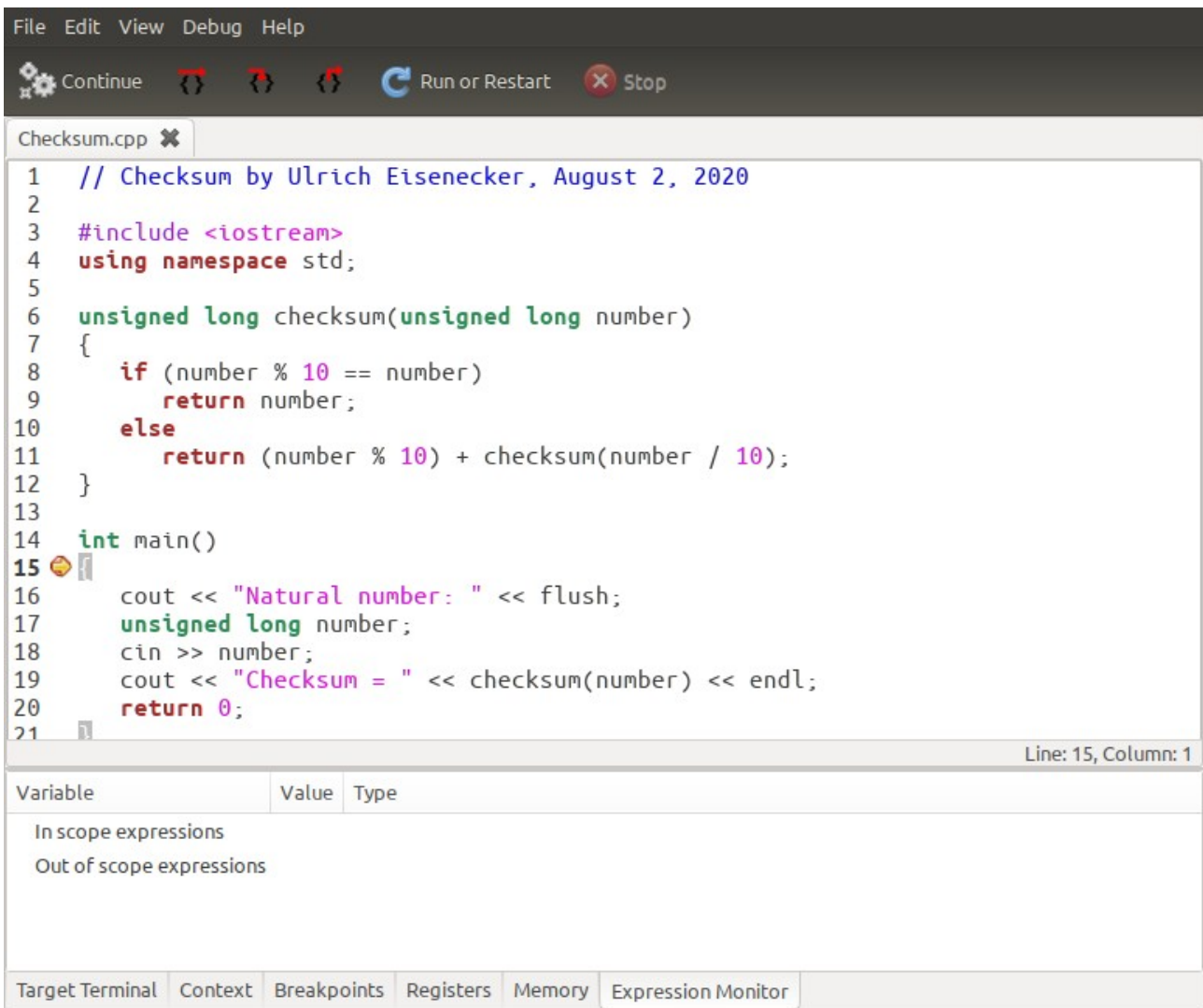


Figure 12: Debugging *Checksum.cpp* with *nemiver*

The upper pane of Figure 12 shows the source code of *Checksum.cpp*. An arrow on a disc points to the opening brace of the body of *main()* (line 15). Actually, these are two overlapping symbols. The first is a red ball indicating a *breakpoint*, which is always automatically set at the beginning of *main()* by *nemiver*. The second is a yellow arrow pointing to the line to be executed next. The *expression monitor* is active in the lower pane. It shows expressions of the actual scope and expressions outside this scope.

There are three icons with red arrows in the toolbar. They correspond to the following debugging actions:

- *Step over*. Executes the next line, executing one or more functions, but not stepping into any function. In this way, a sequence of statements of the same hierarchy level can be stepped over.
- *Step into*. Executes the next line, but step into the function that is executed first. This way it is possible to trace function calls upwards in the *function call stack*. One should step only into functions for which source code is available. Otherwise, the corresponding assembler code will be displayed.
- *Step out*. Exits the active function. Execution continues at the caller level. It is common practice to step into a function, then step over its statements, and finally step out to return to the calling location.

In the toolbar, the icon with the blue round arrow runs or restarts the program. When it is selected, the program is executed. It waits for input and then outputs results. To make inputs or see the output the *Target Terminal* tab, located in the lower pane, must be selected. After that the console window is displayed in the lower pane. It is possible to restart a program at any time.

The icon in the toolbar with the gears continues the execution of the program at the current position of debugging. It starts the program if it is not running yet.

The actions mentioned above are also available from the *debug menu*.

Setting breakpoints is possible only from the debug menu. A *breakpoint* can be set at any statement of the program for which code has been generated. When the program is executed, the debugger suspends the execution immediately before a breakpoint is reached. After that, the programmer can perform actions such as *step over* or *inspect the values of variables*. It is also possible to continue the execution of the program. In this case, the program is executed until the debugger reaches the next breakpoint or the program is terminated. Breakpoints are very useful to debug functions or locations of a program without having to execute many steps manually before reaching the desired location.

To inspect a variable, one simply hovers over it with the mouse. A pop-up message will appear informing about the name, value and type of the variable. A variable can also be permanently included in the expression monitor, which distinguishes between variables that are currently in the scope and those that are not.

Selecting the *Context* tab at the bottom of the window toggles the pane to display the *function call stack* as well as the local variables and function arguments (Figure 13).

Figure 13: Debugger shows function-call stack and local variables

The screenshot shows a debugger window with the following components:

- Menu Bar:** File, Edit, View, Debug, Help
- Toolbar:** Continue, Run or Restart, Stop
- Code Editor:** Displays the source code for `Checksum.cpp`. The code is as follows:


```

1 // Checksum by Ulrich Eisenecker, August 2, 2020
2
3 #include <iostream>
4 using namespace std;
5
6 unsigned long checksum(unsigned long number)
7 {
8     if (number % 10 == number)
9         return number;
10    else
11        return (number % 10) + checksum(number / 10);
12 }
13
14 int main()
15 {
16     cout << "Natural number: " << flush;
17     unsigned long number;
18     cin >> number;
19     cout << "Checksum = " << checksum(number) << endl;
20     return 0;
21 }

```
- Call Stack Table:** Located at the bottom, it shows the current state of the program's execution stack.

Tl	Frame	Function	Arguments	Location	Variable	Value	Type
1	0	checksum	(number = 9)	Checksum.cpp:8	Local Variables		
	1	checksum	(number = 98)	Checksum.cpp:11	Function Arguments		
	2	checksum	(number = 987)	Checksum.cpp:11	number	987654321	unsigned long
	3	checksum	(number = 9876)	Checksum.cpp:11			
	4	checksum	(number = 98765)	Checksum.cpp:11			
	5	checksum	(number = 987654)	Checksum.cpp:11			
	6	checksum	(number = 9876543)	Checksum.cpp:11			
	7	checksum	(number = 98765432)	Checksum.cpp:11			
	8	checksum	(number = 987654321)	Checksum.cpp:11			
	9	main	()	Checksum.cpp:19			
- Bottom Panel:** Includes tabs for Target Terminal, Context, Breakpoints, Registers, Memory, and Expression Monitor.

Selecting *Switch to Assembly* from the debug menu displays the assembler code in the upper pane. Selecting *Switch to Source* from the debug menu displays the source program again.

Now an example debugging session can be described:

1. *Run or Restart.* The execution arrow then points to the opening brace of `main()` (line 15).
2. *Step over.* The yellow arrow now points to the output statement (line 16).
3. *Step over.* The yellow arrow now points to the input statement (line 18). The statement `unsigned long number;` is omitted because no object code was generated for it.

4. *Step over*. The program seems to *hang*. But it doesn't. When the *Target Terminal* tab is selected in the bottom row of the window, the console window is displayed in the bottom pane. In fact, the program is waiting for input. The number 987654321 is entered.
5. Then the lower pane must be changed to *Context*.
6. The arrow now points to the line that calls `checksum()` and outputs its return value (line 19).
7. *Step into*. The lower pane shows in the left part the function call stack and in the right part the local variables and the function arguments.
8. Now *step into* is repeated until the function call stack contains 10 entries from 0 to 9. The first entry (frame 0) shows that `checksum()` is called with `number = 9` as argument. Below that, all previous calls to `checksum()` are listed. This is the result of executing a recursive function that calls itself. A function call consumes memory and time. The memory reserved for the function call stack is limited. Under normal circumstances, recursive calls to `checksum()` will not exhaust the function call stack. But functions with deeper recursion and more and especially large value parameters can cause an overflow of the function call stack, causing the program to crash. Recursion can make for very compact and elegant implementations, but it is not guaranteed to work optimally in C++. This topic will be revisited later.
9. Now, the entry *Switch to Assembly* is selected from the debug menu. The size of the window in Figure 14 has been increased to show the complete assembler code for `checksum()`. Below a line of source code, the assembler code generated for that line is displayed. The first two columns contain positional information, the third column lists the *mnemonic codes* (abbr. *mnemonics*) of the assembler instructions, and the fourth column lists the operands of the assembler instructions, such as memory addresses and processor registers. To give a rough idea of what mnemonics stand for, here are some examples. The mnemonic *mov* means to ***move something to a location***, *cmp* means to ***compare something***, and *jne* means to ***jump if not equal***. The difference between high-level statements in C++ and mnemonics including their operands, is enormous. Of course, it is possible to write very short programs or routines directly in assembler if that is required for performance. But writing larger software in assembler language is an almost impossible task. Moreover, a C++ compiler can perform extensive optimizations that may not be performed by humans. Therefore, it is very difficult to outperform an optimizing compiler by writing assembler code directly.
10. Now the *Switch to Source* entry of the debug menu is selected.
11. Then *step out* is repeated until only `main()` is active in the function call stack.
12. *Continue*. The program is finally terminated.
13. Selecting the *Target Terminal* tab displays the checksum.

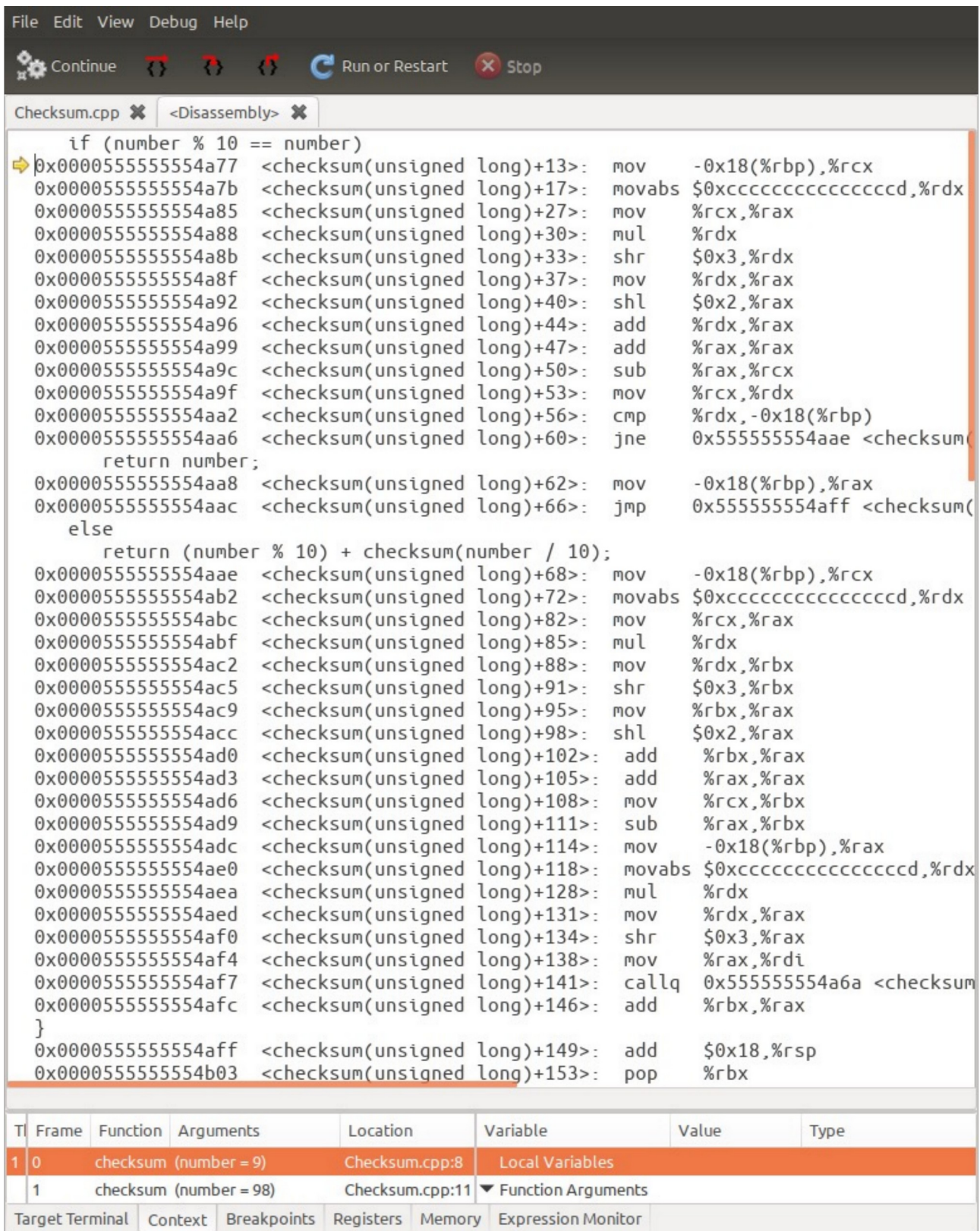


Figure 14: Debugger shows source lines and assembler code

Now the most important functionality for debugging has been introduced. There is more a debugger can do, and there are different debuggers with various graphical frontends. They differ in functionality and especially in the way the functionality is

presented. Therefore, it is pointless to give a more in-depth written tutorial. Learning to debug with a particular debugger requires intensive training in practice.

4. Further Details on Basic Concepts

Now the basic concepts mentioned so far shall be looked at in more detail. Since C++ is a very complex language, this presentation is necessarily incomplete. Many details are omitted and some things are not mentioned at all.

If the content presented in this text is confidently mastered and programming in C++ is to be done in practice, it is highly recommended to consult further sources of information on programming in C++, e.g. (Breymann, 2023) and (Gregoire, 2020).

Reading a book or using it as a reference can be very time consuming. In contrast, online resources can be more up-to-date and provide easy access. Some websites that are useful as references for C++ and provide relevant information are (*Cppreference.Com*, n.d.) and (*Cplusplus.Com - The C++ Resources Network*, n.d.). In addition, many Wikipedia entries provide adequate information on computer science topics.

The C++ standard itself is a less suitable source of information for beginners. It is a densely written technical document that describes the C++ programming language and the libraries included in the standard. The official standard is distributed through the ISO store. At the time of writing, the most recent version (1,853 pages) is available at (ISO/IEC, 2020). An earlier working draft of C++ 17 is (*Working Draft, Standard for Programming Language C++*, n.d.). An unofficial, more recent online version is (Song, n.d.).

4.1. Fundamental Types

C++ has several *fundamental types* built into the language (*Fundamental Types - Cppreference.Com*, n.d.). The most important top-level category of fundamental types is *arithmetic types*. Arithmetic types include *integral* and *floating point types*.

Table 9 provides an overview of the fundamental types, their categories, and equivalents, i.e. alternative type names.

`void` is the only *incomplete type*. Hence it is impossible to have values of type `void`.

`std::nullptr_t` is the type of the null pointer literal, `nullptr`. In fact, `nullptr` is a keyword in C++, but not its type, which is defined in the `<cstdlib>` header.

Type category	Sub category	Sub-sub category	Type	Equivalent(s)
Incomplete			<code>void</code>	
			<code>std::nullptr_t</code>	
Arithmetic	Integral	Boolean	<code>bool</code>	
		Character	<code>char</code> <code>signed char</code> <code>unsigned char</code>	

Type category	Sub category	Sub-sub category	Type	Equivalent(s)
			wchar_t char8_t char16_t char32_t	
		Integral	short int	short signed short signed short int
			unsigned short int	unsigned short
			int	signed signed int
			unsigned int	unsigned
			long int	long signed long signed long int
			unsigned long int	unsigned long
			long long int	long long signed long long signed long long int
			unsigned long long int	unsigned long long
	Floating point		float	
			double	
			long double	

Table 9: Fundamental types in C++

4.1.1. Integral Types

The sub category *Integral* from Table 9 includes the sub-sub categories *Boolean*, *Character*, and *Integral*. They are described below.

4.1.1.1. bool Type

`bool` is the type for logical values. There are two logical values, namely `false` and `true`. Obviously, a single bit would be sufficient to store a boolean value. However, this would be very inefficient with respect to standard computing hardware. Therefore, the size of `bool` is implementation defined, i.e. `bool` occupies one byte or more.

`false` can be automatically converted to the integral value 0, `true` to 1. An integral value of 0 can be automatically converted to `false`, any other integral value can be automatically converted to `true`.

The program in Listing 10 demonstrates the use of the type `bool` and the corresponding operators.

```

1 // Booleans by Ulrich Eisenecker, February 17,2021
2
3 #include <iostream>

```

```

4 using namespace std;
5
6 void printTruthTableNot()
7 {
8     cout << "\nlogical not" << endl;
9     for (bool p : {false,true})
10    {
11        bool r { !p };
12        cout << p
13            << "\t-->\t"
14            << r << endl;
15    }
16 }
17
18 void printTruthTableAnd()
19 {
20     cout << "\nlogical and" << endl;
21     for (auto p : {false,true})
22     {
23         for (auto q : {false,true})
24         {
25             auto r { p && q };
26             cout << p << '\t'
27                 << q << '\t'
28                 << " -->\t"
29                 << r << endl;
30         }
31     }
32 }
33
34 void printTruthTableInclusiveOr()
35 {
36     cout << "\nlogical or" << endl;
37     for (auto p : {false,true})
38     {
39         for (bool q : {false,true})
40         {
41             auto r { p || q };
42             cout << p << '\t'
43                 << q << '\t'
44                 << " -->\t"
45                 << r << endl;
46         }
47     }
48 }
49
50 void printConversionsBoolToInt()
51 {
52     cout << "\nconverting bool to int" << endl;
53     bool p { false };
54     int i { p };
55     cout << "bool value " << p
56         << " becomes int value " << i
57         << endl;
58     auto q { true };
59     i = q;
60     cout << "bool value " << q
61         << " becomes int value " << i
62         << endl;
63 }
64
65 void printConversionsIntToBool()
66 {
67     cout << "\nconverting int to bool" << endl;
68     for (int i { -2 }; i <= 2; ++i)
69     {
70         bool p = i; // calling type conversion constructor

```

```

71     cout << "int value " << i
72         << " becomes bool value " << p
73         << endl;
74 }
75 }
76
77 int main()
78 {
79     // boolalpha causes output of "false" or "true"
80     cout << boolalpha;
81     printTruthTableNot();
82
83     // noboolalpha causes output of 0 or 1
84     cout << noboolalpha;
85     printTruthTableAnd();
86
87     // switching back to "false" or "true"
88     cout << boolalpha;
89     printTruthTableInclusiveOr();
90
91     printConversionsBoolToInt();
92     printConversionsIntToBool();
93 }

```

Listing 10: Booleans.cpp

It has several functions whose names begin with `print`, which are called by the `main()` function. The name of each function expresses the purpose of the function. The first three functions print truth tables for the logical operators applicable to boolean operands. The `!` operator (*logical Not*) accepts only one `bool` operand and negates its value. The `&&` operator combines two `bool` operands according to the *logical And*. The `||` operator combines two `bool` operands according to the *logical Or* (also known as *inclusive Or*).

The other two functions output the effects of converting `bool` values to `int` values and vice versa. When these two functions are executed, they cause exactly the conversions described above.

The `printTruthTableNot()` function introduces a novelty, namely a *range based for loop*. It contains an *initialization statement*, here `bool p`, followed by a colon and a *range expression*, here the curly initialization list `{false, true}`. The *loop statement* can be a single statement or a compound statement. The range based for loop iterates over the given *range*, here `{false, true}`, and binds the *loop variable*, here `p`, to the next element of the range in each iteration. In this way, `p` can be used in the loop statement. Of course, the type of the loop variable and the type of the elements of the range must match. This has been done explicitly here. ***The range based for loop automatically ensures that the range is not exceeded during its execution.***

The `printTruthTableNot()` function introduces another new feature, namely the use of `auto` for defining variables. ***When a variable is defined and initialized, its defining type can be usually inferred from the initializing value. In such cases, the `auto` keyword can be used to define a variable of the inferred type.*** The type

of the loop variable in the initialization statement is inferred from the range expression, which contains `bool` values. The type of `r` is inferred from the result of `p && q`, which is of type `bool`.

The `printTruthTableInclusiveOr()` function uses *type inference* with `auto` and explicit type declaration with `bool`. `auto` is very convenient, but has also some limitations. For example, `auto` **is not suitable when an initializer list is used to initialize an object**. Also, it has some special properties that must be taken into account when using it. They will be explained later.

There is nothing special to say about the `printConversionsBoolToInt()` function. This is different for the `printConversionsIntToBool()` function. It uses a (classic) *for loop*. The header of a `for` loop, enclosed in parentheses, contains three parts separated by semicolons. The first part is an *initialization statement*. It defines the loop variable and initializes it. In `int i { -2 }` the variable `i` is of type `int` and it is initialized with the value `-2`. The second part uses the loop variable in a *condition*. As long as the condition evaluates to `true`, the loop will execute. Here, the condition is `i <= 2`, so the condition evaluates to `true` as long as `i` has a value less than or equal to `2`. The third part *changes the loop variable* so that the condition is finally evaluated as `false` and the `for` loop is terminated. Here, this part is simply `++i`. `++` is the *increment operator*, which increases the value of a variable of an arithmetic type by `1`. It can be placed *before* or *after* the variable. If it stands before the variable, it immediately increases the value of the variable by `1` before it is evaluated (*pre-increment operator*). If it is after the variable, the variable is evaluated first and then incremented by `1` (*post-increment operator*). There is also the `--` operator, which decrements a variable by `1`, also in the form *pre-decrement* and *post-decrement*. The header of the `for` loop is followed by either a *simple statement* or a *compound statement*. Some programmers recommend always using a compound statement. This recommendation is adopted in this text. There are many variations of a `for` loop. Most, if not all, compromise the understandability of a `for` loop. Therefore, only the version of a `for` loop presented above should be used.

In line 70 (function `printConversionsIntToBool()`), the variable `p` is declared and initialized with `bool p = i;`. An initializer list cannot be used here, since this would require an explicit type conversion from `int` to `bool`, which is not performed automatically.

Logical operators are subject to *short-circuit evaluation*. Suppose the functions `p()` and `q()` each return a `bool` value. If the expression `p() && q()` is evaluated and `p()` returns `false`, the entire expression evaluates to `false` without regard to the value returned by `q()`. For this reason, the `q()` function is not called. Another example is the expression `p() || q()`. If `p()` returns `true`, the entire expression is known to be `true` and the function `q()` is not called. So it is not guaranteed that every operand of logical operators is evaluated.

Although `bool` is an arithmetic type, the decrement and increment operators are not available for it. Therefore, the second statement in `bool b { false }; ++b;` is invalid.

4.1.1.2. Character Types

There are several types for characters. The most common one is `char`. Normally a variable of type `char` occupies 1 byte. So it can represent 256 different values. However, it depends on the compiler and the platform whether `char` is signed or unsigned. If it is signed, i.e. `signed char`, its value range is `-128..127`, if it is unsigned, i.e. `unsigned char`, its value range is `0..255`. Although `char` must be either signed or unsigned, `char`, `signed char` and `unsigned char` are treated as different types by the compiler.

Each of these `char` types can store characters encoded in the 7-bit *American Standard Code for Information Interchange* (ASCII for short). ASCII is an early character encoding still widely used today (“ASCII,” 2021). It defines the ASCII codes with values `0..127`, including control characters such as `'\n'` (line feed) or `'\t'` (tab), alphanumeric characters, and some special characters. The eighth bit is unused. The original ASCII does not contain German umlauts and “ß”. Therefore, these characters should not be used in programs or as their input.

Since character types are integral types, all corresponding operators apply to them as well. More about integral operators follows below in connection with the integral types for numbers.

The remaining character types represent either variable length character encodings of `char8_t` for UTF-8) or character encodings with a larger size. More about this can be found in (Breymann, 2023), for example.

4.1.1.3. Integer Types

The integer types – in the strictly numeric sense – can have different sizes and be either signed or unsigned. According to the C++ standard (*Fundamental Types - Cppreference.Com*, n.d.), `short int` occupies at least 16 bits, `signed int` at least 16 bits, `signed long int` at least 32 bits, and `signed long long int` at least 64 bits. The actual size is platform specific. It would be perfectly legal for all of the above types to have a size of 64 bit on a given platform. If `signed` or `unsigned` is omitted, `signed` is assumed by default. There are many synonyms among the names of integral types (last column of Table 9, labelled *Equivalents*). For example, `short int`, `short`, `signed short`, and `signed short int` all denote the same type. It may be a matter of personal taste or coding standard which name is chosen. However, it should be handled consistently throughout the program.

If a program contains an integer literal, its type is deduced from its value. If it fits into `int`, `int` is chosen as type. If it does not fit into `int`, but fits into `long int`, then `long int` is chosen. If its value does not fit in `long int`, then `long long int` is chosen. What happens if its value cannot be represented as `long long int` is – roughly spoken – *implementation defined*. In addition, the suffix `l` or `L` can be used to indicate that the type of an integral literal is `long int`, and the suffix `ll` or `LL` can be used to indicate that the type is `long long int`. For example, `1l` is a literal of type `signed long`, and `2LL` is a literal of type `signed long long`. Only unsigned literals can be typed in source code. For example, if the source code contains `-1L`, it means the positive literal `1` of type `signed long`, to which the unary minus, `-`, is applied to change its sign.

To indicate that a literal has an unsigned integral type, the suffix `u` or `U` is appended. For example, `3lu` and `3UL` are two literals with the value `3` of type `unsigned long`.

Integer literals can be represented in four different bases. A decimal literal (base 10) starts with a non-zero decimal digit (1..9) followed by zero or more decimal digits (0..9), for example `42`. A binary integral (base 2) starts with the prefix `0b` or `0B` followed by at least one binary digit (0, 1), for example, `0b00101010`. An octal literal starts with the digit zero followed by zero or more octal digits (0..7), e.g. `052`. A hexadecimal integral starts with the prefix `0x` or `0X`, followed by one or more hexadecimal digits (0..F), e.g. `0x2A`.

The readability of integral numbers can be improved by inserting one or more apostrophes, for example `4'200'420'0421`. The apostrophes are ignored by the compiler when parsing the literal. ***Apostrophes should be placed from right to left in groups of equal size, e.g. triplets for decimal literals.***

4.1.1.4. `std::byte` Type

The `<cstdint>` header provides the type `std::byte`. It is neither a fundamental type – and therefore not listed in Table 9 – nor an arithmetic type. It is guaranteed to occupy exactly one byte of memory. Only bit operators are defined for `std::byte`. For this reason, `std::byte` should be used whenever it must be guaranteed that an object is exactly 1 byte in size. Whenever this is the case, `std::byte` should be chosen instead of `char`. For this reason, `std::byte` is mentioned here even though it is not a fundamental type.

4.1.1.5. Arithmetic Operators

There are several arithmetic operators that can be applied to integral types. The *unary plus*, `+`, explicitly stands for a positive integral value. If omitted, the integral value is positive by default. The *unary minus*, `-`, inverts the sign of its operand.

The *addition operator*, +, returns the sum of its left and its right operand, e.g., 5 + 3 gives 8. The *subtraction operator*, -, returns the difference by subtracting the second operand from the first operand, e.g., 5 - 3 gives 2. The *multiplication operator*, *, returns the product of its two operands, e.g., 5 * 3 gives 15. The *division operator*, /, returns the result of dividing the first operand by the second operand. If both operands are integers, / performs an integer division, e.g. 5 / 3 results in 1. Any fractional part of the division is truncated. Addition and multiplication are *commutative operators*, subtraction and division are non-commutative. Commutative means, that swapping the two operands of an operator basically does not affect the result of the operation. Non-commutative means that commutativity cannot be guaranteed.

The *modulo operator*, %, returns the remainder of the integer division of the first operand by the second operand, e.g. 5 % 3 gives 2.

There are also six bitwise operators (*Arithmetic Operators - Cppreference.Com*, n.d.):

1. The *bitwise-Not operator* (also called *Negation*), ~, is a unary operator that inverts all bits of its integral operand, for example,


```
~0b0000'1111'0000'1111
0b1111'0000'1111'0000.
```
2. The *bitwise-And operator*, &, performs an And for each bit of its two integral operands, for example,


```
0b1111'1111'0000'0000
& 0b1010'1010'1010'1010
0b1010'1010'0000'0000.
```
3. The *bitwise-Or operator*, |, performs an (inclusive) Or for each bit of its two integral operands, for example,


```
0b1111'1111'0000'0000
| 0b1010'1010'1010'1010
0b1111'1111'1010'1010.
```
4. The *bitwise-Xor operator*, ^, performs an exclusive Or for each bit of its two integral operands, for example,


```
0b1111'1111'0000'0000
^ 0b1010'1010'1010'1010
0b0101'0101'1010'1010.
```
5. The *bitwise-left shift operator*, <<, shifts all bits of the first operand to the left by as many bits as the second operand specifies, for example,


```
0b1111'1111'0000'0000 << 2
0b1111'1100'0000'0000.
```

Any bit shifted before the MSB is lost. Zeroes are shifted in from the right.
6. The *bitwise-right shift operator*, >>, shifts all bits of the first operand to the right by as many bits as the second operand specifies, for example,


```
0b1111'1111'0000'0000 >> 2
0b0011'1111'1100'0000.
```

Each bit that is shifted behind the LSB is lost. Zeroes are shifted in from the left.

4.1.1.6. Promotions and Conversions

There are plenty of *promotions* for integral types (*Implicit Conversions - Cppreference.Com*, n.d.). The purpose of a promotion is to choose a target type that can correctly represent the value of an integral expression. Apart from promotions, *conversions* between integral types can change values and potentially lose precision. Therefore, it should be ensured that the values of integral literals and variables, as well as expressions that use integrals, can always be correctly represented by their target type.

Since an arithmetic overflow is handled differently for signed and unsigned integrals, signed and unsigned integers should not be mixed in arithmetic expressions.

The program in Listing 11 demonstrates the arithmetic overflow of signed and unsigned integral types and points out a peculiarity of applying bit-shift operators to signed integral types.

```
1 // UnsignedVsSigned by Ulrich Eisenecker, March 1, 2021
2
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     // underflow and overflow for unsigned
9     cout << "unsigned" << endl;
10    unsigned u { 0 };
11    cout << "lower bound:      " << u << endl;
12    u = u - 1;
13    cout << "lower bound - 1: " << u << endl;
14    u = 0;
15    u = ~u; // negate all bits of u
16    cout << "upper bound:      " << u << endl;
17    u = u + 1;
18    cout << "upper bound + 1: " << u << endl;
19
20    // underflow and overflow for signed
21    cout << "signed" << endl;
22    u = 0; u = ~u; u = u >> 1; u = ~u;
23    signed s = u; // calling type conversion constructor
24    cout << "lower bound:      " << s << endl;
25    s = s - 1;
26    cout << "lower bound - 1: " << s << endl;
27    u = 0; u = ~u; u = u >> 1;
28    s = u;
29    cout << "upper bound:      " << s << endl;
30    s = s + 1;
31    cout << "upper bound + 1: " << s << endl;
32 }
```

Listing 11: UnsignedVsSigned.cpp

In line 10 of Listing 11, the variable `u` of type `unsigned` (or `unsigned int`) is declared and initialized with 0 so that all bits of `u` are unset. 0 is the lower bound of unsigned integral types. Subtracting 1 from 0 causes all bits of the unsigned value to be set. This gives the upper bound of the corresponding integral type. If `unsigned` has a size of 32 bit on a given platform, this value is 4,294,967,295, which equals to $2^{32} - 1$.

Next, all bits are set using a slightly more complicated procedure. First, `u` is assigned 0, which resets all bits, and the result is assigned to `u`. Then, the operator *bitwise Not* is applied to `u`, inverting all bits. Since all bits were unset before, all bits are set afterwards. The result is assigned to `u`. The output of `u` again shows the upper bound for this unsigned type.

Adding 1 to the upper bound gives 0, i.e. all bits are unset again.

Line 22 of Listing 11 shows how to set the MSB of an unsigned integral variable and to reset all other bits. First, `u` is assigned 0, which resets all bits. Then, all bits of `u` are negated one bit at a time and the result is assigned to `u`. Then all bits of `u` are shifted to the right by exactly one position. Thereby the LSB is shifted out to the right. A zero bit is shifted in from the left, which now has the value of the MSB. Finally, the bits of `u` are negated again and assigned to `u`. This results in the MSB being set while the remaining bits are unset. This value represents the largest negative value, i.e. the lower bound that a signed variable can contain, but here in an unsigned variable.

In the next line, i.e. line 23, the signed variable `s` is declared and initialized with the value of `u`. This is perfectly legal and does not even lead to a warning (*Implicit Conversions - Cppreference.Com*, n.d.). On a platform which uses 32 bits for signed `int`, this results in the output of -2,147,483,648, which is equivalent to $-2^{32}/2$. As mentioned earlier, initialization with an initializer list is not possible here because the variables `s` and `u` are of different types.

Subtracting 1 from this value, all bits are negated. Now the MSB is unset, while all other bits are set. The result is the upper bound of signed `int`. If signed `int` occupies 32 bit, this corresponds to $2^{32}/2 - 1$, which gives 2,147,483,647.

Then `u` is used to create a bit pattern in which the MSB is unset and all other bits set. Now the bits of `u` represent the upper bound of signed `int`. Then `s` is assigned the value of `u`. When sent to standard output, the same value as before is output.

Adding 1 to `s` and reassigning the result to `s` flips all bits of `s`, resulting in the lower bound introduced above.

Why are the bit operations for creating the bit pattern for lower and upper bound of signed `int` not directly applied to variable `s`? The reason is that bit-shift operators behave differently for signed `int` (*Arithmetic Operators - Cppreference.Com*, n.d.). Exactly how they behave exactly is partly undefined and partly determined by the

implementation. Therefore, *the use of bit shift-operators for signed int is not recommended*.

While arithmetic overflow is well defined for unsigned integral types, it leads to undefined behavior for signed integral types (*Arithmetic Operators - Cppreference.Com*, n.d.). Obviously, an arithmetic overflow does not lead to a runtime error for signed integral types. It should be mentioned that the program in Listing 11 may produce different output when compiled by different compilers.

4.1.1.7. Assignment Operators

An expression of the form $x = x \otimes y$, where \otimes stands for an arithmetic or a bit operator, can be rewritten as $x \otimes= y$. All operators of the form $\otimes=$ belong to the class of *assignment operators* (*Assignment Operators - Cppreference.Com*, n.d.). For example, $u = u >> 1;$ becomes $u >>= 1;$. Of course, this scheme applies only to operators \otimes that have two operands (the number of operands an operator has is called *arity*). For example, operator \sim in $u = \sim u;$ has only one operand. Thus, it cannot be rewritten in this way. The program in Listing 12 shows how to rewrite the program in Listing 11 using the assignment operators introduced earlier.

```
1 // UnsignedVsSignedShorter by Ulrich Eisenecker, March 3, 2021
2
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     // underflow and overflow for unsigned
9     cout << "unsigned" << endl;
10    unsigned u { 0 };
11    cout << "lower bound:      " << u << endl;
12    u -= 1;
13    cout << "lower bound - 1: " << u << endl;
14    u = 0;
15    u = ~u; // negate all bits of u
16    cout << "upper bound:      " << u << endl;
17    u += 1;
18    cout << "upper bound + 1: " << u << endl;
19
20    // underflow and overflow for signed
21    cout << "signed" << endl;
22    u = 0; u = ~u; u >>= 1; u = ~u;
23    signed s = u; // calling type conversion constructor
24    cout << "lower bound:      " << s << endl;
25    s = s - 1;
26    cout << "lower bound - 1: " << s << endl;
27    u = 0; u = ~u; u >>= 1;
28    s = u;
29    cout << "upper bound:      " << s << endl;
30    s += 1;
31    cout << "upper bound + 1: " << s << endl;
32 }
```

Listing 12: *UnsignedVsSignedIntShorter.cpp*

4.1.1.8. Floating Point Types

There are three types for representing *floating-point* values.

1. `float` is the single precision floating point type,
2. `double` is the double precision floating point type, and
3. `long double` is the extended precision floating point type.

Usually the compiler generates code for double precision arithmetic, even if only `float` values are involved. Therefore, `float` values are converted to double values before arithmetic operations such as addition or division are performed. Also, `long double` should be more precise than `double`, but it is possible, that it has the same properties as `double` (again, `double` and `long double` are different types). Therefore, it is recommended to use `double` as floating point type by default, unless memory or precision is crucial.

To understand floating point types, knowledge of *scientific notation* for real numbers is helpful. Here is an example of a floating point number in scientific notation:

`-1.2345e-2`

In scientific notation the sign indicates whether the number is positive or negative. In the example above, the sign is highlighted in red (-). A negative sign is mandatory, a positive sign can be omitted. The sign is followed by the mantissa, also called the *significand*, which is highlighted in green (1.2345). The mantissa shows exactly one non-zero digit which is optionally followed by a decimal point and other digits, but no trailing zero digit. Then the *exponent*, usually assumed to be in base ten, indicates by how many digits the mantissa must be shifted. The exponent is highlighted in blue (e-2). In scientific notation *e* or *E* has the meaning of 10^{exponent} , i.e. the preceding number has to be multiplied by the power of ten of the following exponent. A positive exponent means that the decimal point in the mantissa must be shifted to the right by *exponent* digits, a negative exponent means, that it has to be shifted to the left by *exponent* digits.

Similar to scientific notation, each floating-point type usually includes exactly one sign bit, several bits for the exponent, and the bits for the mantissa. The number of bits for the exponent determines the range of a floating-point type, and the number of bits used for the mantissa determines its precision.

Floating-point literals can be written in different ways (*Floating Point Literal - Cppreference.Com*, n.d.). In most cases, it is sufficient to use either the plain decimal representation or the scientific decimal representation. If the literal is written without suffix, e.g. `3.12` or `3.12e+0`, it is of type `double`. The suffix `f` or `F` causes the literal to be of type `float`, e.g. `3.f` or `3.0e+0F`. The suffix `l` or `L` indicates that the

literal is of type `long double`, e.g. `2.71828L` or `2.71828e01` (the positive sign of the exponent can be omitted).

Any floating-point value can be converted into another floating-point value (*Implicit Conversions - Cppreference.Com*, n.d.). **If the exact value is preserved, it is a promotion, not a conversion.** For example, a float value is promoted to a double value.

Any floating-point value can be converted into an integer value. During the conversion, the fractional part of the floating-point value is truncated. If the floating-point value does not fit into the integral type, the behavior is undefined.

Any integer value can be converted into a floating-point value. If the integral value does not fit into the floating-point type, the behavior is undefined.

Since failed conversions lead to undefined behavior, it is difficult to give valid instances for these consequences. What happens, when such a conversion fails depends on the hardware and the compiler. One way to assign a huge `long double` value to a variable of type `short` is to have the integral variable to take the largest possible value that can be represented by `short`. Assigning the largest possible unsigned `long long` value to a variable of type `float` may result in a loss of precision. But all this is speculative, since the behavior in these cases is undefined.

Various manipulators control the format when sending floating point values to the output (*Input/Output Manipulators - Cppreference.Com*, n.d.). The program in Listing 13 demonstrates some of them.

```
1 // FloatOutput by Ulrich Eisenecker, November 2,2021
2
3 #include <iostream> // implicitly includes <ios>
4 #include <iomanip> // because of setprecision
5 using namespace std;
6
7 int main()
8 {
9     double d { };
10    cout << "Formatting floating point values\n"
11         << "Enter 42.0 to terminate program.\n"
12         << endl;
13    do
14    {
15        cin >> d;
16        cout << "default: " << d << endl
17             << "scientific: "
18             << scientific << d << endl
19             << "fixed: "
20             << fixed << d << endl
21             << "setprecision(20): "
22             << setprecision(20)
23             << d << endl
24             << "hexfloat: "
25             << hexfloat << d << endl
26             << "defaultfloat: "
27             << defaultfloat << d << endl;
28    } while (d != 42.0);
29 }
```

The sample dialog shown in Figure 15 shows the result of running this program. It may slightly vary on another computer.

```
Formatting floating point values
Enter 42 to terminate program.

5
default: 5
scientific: 5.000000e+00
fixed: 5.000000
setprecision(20): 5.00000000000000000000
hexfloat: 0x1.4p+2
defaultfloat: 5
42
default: 42
scientific: 4.20000000000000000000e+01
fixed: 42.00000000000000000000
setprecision(20): 42.00000000000000000000
hexfloat: 0x1.5p+5
defaultfloat: 42
```

Figure 15: Sample dialog for executing `FloatOutput.cpp` (Listing 13)

The number 5 is a perfect input, even it is not explicitly entered as a floating-point value, i.e. 5.0. Without further precautions the output is 5. The manipulator `scientific` causes the output to be in scientific format and to consist of a mantissa and an exponent. The exponent is positive and zero because the decimal point does not need to be moved. `fixed` causes a fixed number of digits to be used for the output. `setprecision(int_val)` causes the output to use `int_val` digits after the decimal point. After `hexfloat` the floating point value is output in hexadecimal format. The default format for floating-point output can be restored with `defaultfloat` – at least almost. As the second run of the do loop shows, the number of digits used for output is still set to 20.

It is very instructive to run this program with different numbers. First, the format for the number input can be varied, e.g. by using a decimal point or scientific notation. Second, the number itself can be varied, e.g. numbers with large mantissas or negative or positive exponents of different sizes.

Entering an invalid value has a noticeable effect. The program no longer asks for the input, but starts to output the same value, mostly 0 in different formats, again and again. The program must be interrupted by pressing `Ctrl-C`, i.e. press the *Control* key, hold it and additionally press the *c* key. The reason for this is that after an invalid input the object `cin` is in an error state and does not accept any further inputs. The different states of `cin` and other streams will be discussed later.

It must be understood that floating-point numbers are an imperfect substitute for *real numbers* (“List of Types of Numbers,” 2021). The set of real numbers includes both the set of *rational numbers*, which is *countably infinite*, and the set of *irrational*

numbers, which is not *countably infinite*. Because of the inclusion of the irrational numbers, the set of real numbers is also not countably infinite. Unlike the real numbers, all floating-point numbers are finite (and therefore countable) because they are represented with a limited number of bits. Theoretically, all floating-point numbers could be enumerated by systematically creating all 2^n permutations (“Permutation,” 2021) of the n bits used to represent a floating-point type. A consequence of this is, that calculations with floating-point numbers are not *principally exact*. *Principally exact* means, that every calculation with floating-point numbers gives an exact result – and this is not the case. Of course, there are some calculations with floating-point numbers that give an exact result. For example, the expression $0.4 + 0.4$ gives exactly 0.8, but $0.4 + 0.4 + 0.4$ does not give exactly 1.2! That is demonstrated below using two for loops. The for loop in Listing 14 gives the expected result, which is shown in Figure 16.

```
1 for (double i { 0.0 }; i <= 0.8; i += 0.4)
2 {
3     cout << i << endl;
4 }
```

Listing 14: for loop – exact computation

```
0
0.4
0.8
```

Figure 16: Output generated by for loop – exact computation (Listing 14)

The for loop in Listing 15 leads to the unexpected result shown in Figure 17. Obviously, the last value, 1.2, is missing!

```
1 for (double i { 0.0 }; i <= 1.2; i += 0.4)
2 {
3     cout << i << endl;
4 }
```

Listing 15: for loop – inexact computation

```
0
0.4
0.8
```

Figure 17: Output generated by for loop – inexact computation (Listing 15)

This is because some floating point numbers can be represented exactly internally, but others cannot. This is due to the binary representation of floating point numbers in the decimal system. Excellent background information, including a discussion of how this problem can be solved by choosing a different base, can be found in (Cheng, 2017).

This has several consequences:

- Floating point calculations can generally provide approximative results; there is no guarantee of exact results.

- Results of floating-point calculations should never be checked for equality with a particular value using the `==` operator, but rather to see if the result falls within a certain range. An alternative is to check whether the amount of the difference of the actual and expected value is less than a relatively small value close to zero. The program shown in Listing 16 illustrates both possibilities.
- Important mathematical properties, such as *associativity* or *distributivity*, can be violated in floating point arithmetic. Therefore, floating point calculations should not depend on these properties.
- The use of floating point variables as loop variables should be avoided.

The first output statement of the program in Listing 16 shows again that the result of the calculation of $0.0 + 0.4 + 0.4 + 0.4$ is not exact. Therefore, `d == 1.2` is evaluated as `false`. The second output statement tests whether `d` lies in the interval $(1.1999, 1.2001)$.

```

1 // TestingFloats by Ulrich Eisenecker, November 3 ,2021
2
3 #include <iostream>
4 #include <cmath> // because of fabs()
5 using namespace std;
6
7 int main()
8 {
9     double d { 0.4 + 0.4 + 0.4 };
10    cout << boolalpha
11         << "d == 1.2: " << (d == 1.2) << endl;
12    cout << "1.1999 < d && d < 1.2001: " << (1.1999 < d && d < 1.2001) << endl;
13    cout << "fabs(d - 1.2) < 0.0002: " << (fabs(d - 1.2) < 0.0002) << endl;
14 }

```

Listing 16: `TestingFloats.cpp`

It is worth mentioning that when an interval is specified textually – not in a program – a parentheses is used to indicate an excluded boundary, while the use of a bracket means that the corresponding boundary is included. Thus, $(1.1999, 1.2001]$ specifies an interval where the lower bound is excluded while the upper bound is included. $[1.1999, 1.2001)$ specifies an interval where the lower bound is included but the upper bound is excluded.

In the third output statement the expected value is subtracted from the calculated value. The function `fabs()` (which means *float absolute value*) calculates the absolute value of its argument. Finally, it is checked if this value is smaller than a relatively small value close to zero.

The program in Listing 17 illustrates two other interesting aspects of floating-point arithmetic, namely *infinitely large* results and results that are *not a number*.

```

1 // Infinity_NaN by Ulrich Eisenecker, November 3, 2021
2
3 #include <iostream>
4 #include <cmath> // because of sqrt() and macros INFINITY and NAN
5 using namespace std;

```

```

6
7 int main()
8 {
9     double d { 1.0 / 0.0 };
10    cout << d << endl;
11    cout << boolalpha
12         << "d == INFINITY: " << (d == INFINITY) << endl;
13    d = sqrt(-1.0);
14    cout << d << endl;
15    cout << "d == NAN: " << (d == NAN) << endl;
16    cout << "NAN == NAN: " << (NAN == NAN) << endl
17         << "NAN != NAN: " << (NAN != NAN) << endl;
18 }

```

Listing 17: `Infinity_NaN.cpp`

When `d` is declared, it is initialized with the value of the expression `1.0 / 0.0`. One possibility is that a division by zero causes a runtime error. This is not the case here. Another possibility is that dividing 1.0 by zero results in an infinitely large number, i.e. *infinity*. Although floating-point types cannot represent an infinite number of values, they can very well represent infinity. And this is exactly what happens. The following output statements output *inf*. There is even a macro that defines the internal representation of infinity. Both aspects, the *human readable value inf* and the *internal representation of infinity*, must be distinguished. Evaluating `d == INFINITY` yields true.

Then the square root of -1 is calculated with the `sqrt()` function and the result is assigned to `d`. The calculation of the square root of a negative number results in a complex number. However, complex numbers are not one of the fundamental types of C++. They are defined in the `<complex>` header, but the `sqrt()` function does not know about this header. Another option is to generate a runtime error. But this option is not chosen here, the result rather takes the value *not a number*, NAN for short, which means that the result is invalid. Sending NAN to `cout` prints *nan* in the console window. The next three statements illustrate an interesting property of NAN. When NAN is tested for equality with itself, the result is false. At least, testing NAN for inequality with itself yields true. For more on INFINITY and NAN, see (*INFINITY - Cppreference.Com*, n.d.) and (*NAN - Cppreference.Com*, n.d.).

The `isnan()` function defined in the `<cmath>` header returns true if the passed value is not a number, and false otherwise (*Std::Isnan - Cppreference.Com*, n.d.). The `isinf()` function defined in the `<cmath>` header returns true if the passed value is positive or negative infinity, and false otherwise (*Std::Isinf - Cppreference.Com*, n.d.).

4.1.1.9. Real World and Computer – Again

The importance of the relationship between a problem in the real world and its solution in form of a program has already been emphasized several times, for example in the Section [Program and Reality](#).

This is to be taken up here once again. Integral numbers and real numbers are part of reality, even if they belong exclusively to the immaterial world. They are deeply understood, they are well formalized, and there is a wealth of knowledge and easily accessible documentation about them. If there is a problem involving integer or real numbers, there may already be a solution that just needs to be found, or it may be relatively easy to develop such a solution using the available body of knowledge. The next step is then to transfer this solution to the domain of computers and programming languages. It is now clear that computers and programming languages offer only limited possibilities for implementing this solution. Several limitations must be considered, such as the limited range of values, the limited precision, and the lack of properties that hold in the real world, such as associativity and distributivity. Just think again of the infinite number of real numbers mapped to a single floating point value! One of the reasons for this is that a computer is never a perfect Turing machine. It does not have infinite memory, and the time to perform computations is also limited. Therefore, solutions in the form of programs are often only well thought-out approximations and fine-tuned abstractions of reality or, more generally, of the problem to be solved. ***Therefore, a proper understanding of the requirements associated with the problem is critical. Important requirements should not be overlooked,*** and requirements should not be contradictory. In addition, one should be aware of the limitations that computer hardware, programming languages, and programming paradigms impose on the development and implementation of a solution in the form of software. A basic assumption should also be mentioned. Reality is much more comprehensive and detailed than software can be. Thus, if a problem from reality is mapped to a solution in software with its strong constraints, this will usually be a unidirectional mapping from the problem in reality to the software solution. Reverse inferences from findings in software must therefore always be thoroughly validated. And the more serious the consequences can be, the more intensively and critically these findings should be evaluated.

4.2. More About Types

The following sections explain how to get information about types, introduce type qualifiers, give an insight into the memory organization of a C++ program, take a look at pointers and present some advice on naming identifiers.

4.2.1. Information About Types

Instead of defining the exact properties of a type, C++ specifies exactly how the properties of a type can vary depending on the platform and the compiler. The

question now is *how to retrieve information about the properties of a type*. Fortunately, C++ provides several mechanisms for retrieving such information.

The `sizeof` operator determines how many bytes a type occupies in memory (*Sizeof Operator - Cppreference.Com*, n.d.). The language usage is imprecise here, since only *exemplars* or, in other words, *instances of types* occupy memory, but not the type itself.

`sizeof` can be applied to expressions, including literals, and types, except for – among others – incomplete types. `sizeof` returns a result of type `std::size_t`. This is a type alias for an integer type which can represent the maximum size of objects in bytes (*Std::Size_t - Cppreference.Com*, n.d.). On most platforms, a byte consists of 8 bits. However, this can also vary. The `CHAR_BIT` macro (defined in the `<climits>` header) determines how many bits a byte has on the respective platform (*C Numeric Limits Interface - Cppreference.Com*, n.d.).

The program shown in Listing 18 illustrates the use of the `sizeof` operator.

```
1 // Sizeof by Ulrich Eisenecker, November 3: ,2021
2
3 #include <iostream>
4 #include <climits> // because of CHAR_BIT
5 using namespace std;
6
7 int main()
8 {
9     cout << "A byte has " << CHAR_BIT
10         << " bit on this platform."
11         << endl;
12     // cout << "sizeof(void); " << sizeof(void) << endl;
13     // uncommenting previous line causes an error,
14     // because void is an incomplete type
15     cout << "sizeof(-.0e-42f): " << sizeof(-.0e-42f) << endl;
16     cout << "sizeof(1/0.0): " << sizeof(1 / 0.0) << endl;
17     cout << "sizeof(long double): " << sizeof(long double) << endl;
18     size_t size { sizeof(size_t) };
19     cout << "sizeof(size): " << sizeof(size) << endl;
20 }
```

Listing 18: `Sizeof.cpp`

Figure 18 shows the output generated after compiling and running this program. It may look different on a different computer.

```
A byte has 8 bits on this platform.
sizeof(-.0e-42f): 4
sizeof(1/0.0): 8
sizeof(long double): 16
sizeof(size): 8
```

Figure 18: Output of the `Sizeof.cpp` program (Listing 18)

`sizeof(void)` is illegal, because `void` is an incomplete type. Therefore, this line is commented out and produces no output. On the platform used, a byte has eight bits. The next three statements output the number of bytes occupied by the types `float`, `double`, and `long double` on this platform. First, `sizeof` is applied to a literal (more

precisely, a *literal expression*). Then, `sizeof` is applied to an expression whose two parts are literal expressions. Normally, the compiler would evaluate this expression, since this is possible at compile time. However, when `sizeof` is applied to an expression, the expression is *not* evaluated. Third, `sizeof` is applied to a type. Finally, `sizeof` is used to determine the number of bytes occupied by an object of type `std::size_t`. On this platform, this is 8 bytes, which means that a maximum size of 2^{32} bytes can be represented.

The C++ standard contains a *type support library*. It provides a wealth of options for retrieving type-related information. A particular type-related information, i.e. a property of a type, is called a *trait*. Since a type is information about an expression or a variable, information about types is *meta information*. In the following, a small selection of meta information for types and the access to this information is presented.

The program shown in Listing 19 demonstrates various checks using fundamental types.

```
1 // TypeInformation by Ulrich Eisenecker, August 12, 2024
2
3 #include <iostream>
4 #include <type_traits> // Because of is_arithmetic_v<>, is_integral_v<>,
5 // is_unsigned_v<>, is_signed_v<>,
6 // is_floating_point_v<>, is_same_v<>.
7 #include <cstdint> // Because of std::nullptr_t.
8 using namespace std;
9
10 int main()
11 {
12     cout << boolalpha
13         << "is_arithmetic<>"
14         << endl
15         << is_arithmetic_v<bool> << endl
16         << is_arithmetic_v<char> << endl
17         << is_arithmetic_v<int> << endl
18         << is_arithmetic_v<double> << endl
19         << is_arithmetic_v<nullptr_t> << endl;
20
21
22     cout << "\nis_integral_v<>" << endl
23         << is_integral_v<bool> << endl
24         << is_integral_v<char> << endl
25         << is_integral_v<int> << endl
26         << is_integral_v<double> << endl
27         << is_integral_v<nullptr_t> << endl;
28
29     cout << "\nis_unsigned_v<>" << endl
30         << is_unsigned_v<bool> << endl
31         << is_unsigned_v<char> << endl
32         << is_unsigned_v<int> << endl
33         << is_unsigned_v<double> << endl
34         << is_unsigned_v<nullptr_t> << endl;
35
36     cout << "\nis_signed_v<>" << endl
37         << is_signed_v<bool> << endl
38         << is_signed_v<char> << endl
39         << is_signed_v<int> << endl
40         << is_signed_v<double> << endl
41         << is_signed_v<nullptr_t> << endl;
```

```

42
43     cout << "\nis_floating_point_v<>" << endl
44         << is_floating_point_v<bool> << endl
45         << is_floating_point_v<char> << endl
46         << is_floating_point_v<int> << endl
47         << is_floating_point_v<double> << endl
48         << is_floating_point_v<nullptr_t> << endl;
49
50     cout << "\nis_same_v<>" << endl
51         << is_same_v<bool,bool> << endl
52         << is_same_v<bool,int> << endl;
53
54     cout << "\nis_same_v<>" << endl
55         << is_same_v<int,signed int> << endl;
56 }

```

Listing 19: *TypeInformation.cpp*

`is_arithmetic_v<>` is a so-called *variable template* to which the result of the execution of a so-called *template meta-function* is assigned at compile time, e.g. `is_arithmetic_v = is_arithmetic<T>::value` (a syntactically incorrect simplification that serves to outline the idea).

Consequently, `is_arithmetic_v<bool>` is true because `bool` is an arithmetic type. `is_arithmetic_v<nullptr_t>` is false because `std::nullptr_t` is not an arithmetic type. The remaining template variables do what their names suggest. To use `_v` as a shorthand for `::value` for all C++ traits is possible since C++20. For more information, see (Breymann, 2023), p. 525.

The variable template `is_same_v<>` takes two types as arguments. The result of checking `bool` and `bool` for sameness is obvious. Less obvious, but still correct, is that `is_same_v<int,signed int>` is also true.

At the moment, it is not important to understand exactly what a template is. It is sufficient to use it. To identify a template, be it a variable or function template as before or a class template that will appear later, a pair of angle brackets is appended to its name. A little more information on templates is given in Section [Data](#).

Figure 19 shows the output of a modified version of the program in Listing 19. For a better overview the information is presented in tabular form. Using `is_same_v<>` has been omitted. The listing of the corresponding program is not included here.

	bool	char	int	double	nullptr_t
<code>is_arithmetic_v</code>	true	true	true	true	false
<code>is_integral_v</code>	true	true	true	false	false
<code>is_unsigned_v</code>	true	false	false	false	false
<code>is_signed_v</code>	false	true	true	true	false
<code>is_floating_point_v</code>	false	false	false	true	false

Figure 19: *Type information in tabular form*

The values of `is_unsigned_v<>` and `is_signed_v<>` for `char` may differ on other platforms. The other values should be the same.

Of course, `is_signed_v<double>` is true, because every floating point type is signed. For `std::nullptr_t`, all values are false. Since `std::nullptr_t` is not an arithmetic type, the arithmetic-related information is not applicable and therefore false. Like the other types for which meta information was determined, `std::nullptr_t` is a fundamental type. Therefore, the value of `is_fundamental<>` had to be true for all types listed in the table, including `std::nullptr_t` if a corresponding line was included in the table.

One remarkable observation deserves to be mentioned. When running the modified program under *macOS* compiled with *Apple clang version 18.1.8*, a strange problem occurs: The output width set with the corresponding manipulator is sometimes wrong for false. Therefore, the table is distorted. Compiling and running exactly the same program using *g++ 9.3.0* on a *Linux* platform (and also under *macOS* compiled with *g++ 14.1.0*) gives the expected result. According to a personal communication with Uli Breyman, the problem is caused by the library used (*libc++* on *macOS* and *libstdc* on *Linux*). The reason is apparently an insufficient specification in the C++ standard (*C++ Standard Library Active Issues List*, n.d.).

Finally, some meta information for arithmetic types from the `<limits>` header (*Std::Numeric_limits - Cppreference.Com*, n.d.) is presented. This time, a template called `numeric_limits<>` is used to access information for a type passed as an argument. For example, `numeric_limits<T>::is_specialized` tells whether there is any information at all for the type `T` specified inside the angle brackets. The program shown in Listing 20 contains some usage examples.

```
1 // NumericLimits by Ulrich Eisenecker, March 15, 2021
2
3 #include <iostream>
4 #include <limits> // Because of numeric_limits<>.
5 #include <cstdlib> // Because of std::nullptr_t.
6 using namespace std;
7
8 int main()
9 {
10  cout << boolalpha;
11
12  cout << "nullptr_t" << endl
13       << numeric_limits<nullptr_t>::is_specialized << endl
14       << numeric_limits<nullptr_t>::is_exact << endl
15       << numeric_limits<nullptr_t>::min() << endl
16       << numeric_limits<nullptr_t>::max() << endl
17       << numeric_limits<nullptr_t>::epsilon() << endl;
18
19  cout << "\nint" << endl
20       << numeric_limits<int>::is_specialized << endl
21       << numeric_limits<int>::is_exact << endl
22       << numeric_limits<int>::lowest() << endl
23       << numeric_limits<int>::min() << endl
24       << numeric_limits<int>::max() << endl
25       << numeric_limits<int>::epsilon() << endl;
26
```



```

27     cout << "\ndouble" << endl
28         << numeric_limits<double>::is_specialized << endl
29         << numeric_limits<double>::is_exact << endl
30         << numeric_limits<double>::lowest() << endl
31         << numeric_limits<double>::min() << endl
32         << numeric_limits<double>::max() << endl
33         << numeric_limits<double>::epsilon() << endl;
34 }

```

Listing 20: *NumericLimits.cpp*

`std::nullptr_t` is a fundamental type, but not an arithmetic type. Therefore, it is a good candidate for testing `numeric_limits<nullptr_t>::is_specialized`. Of course, this meta information is false for `std::nullptr_t`. For this reason, it is useless to access the following meta information. Nevertheless, this is done out of pure curiosity. That `numeric_limits<nullptr_t>::is_exact` is false can be seen as an indicator that exactness is not relevant for a type for which `numeric_limits<>` provides no information. The next three pieces of meta information are retrieved by executing so-called *static member-functions*. This can be recognized by the double colon before the identifier and the following pair of matching parentheses. Static member-functions will be explained later. The only issue here is how to use them. `numeric_limits<nullptr_t>::min()` returns the single value that `std::nullptr_t` has, namely `nullptr`. Despite the fact that `numeric_limits<>` does not provide meta information for `std::nullptr_t`, this is a precise information. `numeric_limits<nullptr_t>::max()` behaves exactly the same way. `numeric_limits<>::epsilon()` returns the *machine epsilon*, which is the difference between 1.0 and the next value that can be represented by a given floating-point type. Of course, the value `nullptr` is not meaningful, but nevertheless, the actual result is what one can expect for `std::nullptr_t`.

`numeric_limits<>` provides meta information for `int`. `int` is an exact type, and its lowest, minimum, and maximum values can be easily accessed with `numeric_limits<int>::lowest()`, `numeric_limits<int>::min()`, and `numeric_limits<int>::max()`. These special values do not have to be constructed in a such complicated way as shown in Listing 11. The lowest and the minimum value coincide in the case of `int`. For integer types, there is no suitable definition of epsilon. Therefore, the result 0 returned by `numeric_limits<int>::epsilon()` should not be interpreted.

For `double` all the previously mentioned meta information is available and useful. Therefore, `numeric_limits<double>::is_specialized` returns true. A floating point type does not allow exact calculations. Therefore, `numeric_limits<double>::is_exact` is false. The lowest value is actually the most negative value that can be represented by `double`. The minimum value is the smallest value that can be represented. The lowest, minimum and maximum values may vary on another platform, for example, because `double` is represented with a different number of bytes. The program in Listing 20 outputs only a small portion of the

mantissa of values. To adjust the number of digits of the mantissa, the `setprecision()` manipulator from `<iomanip>` header can be used (see Section [Floating Point Types](#)).

4.2.2. Pointers

When a program is started, the *loader* loads it into the computer's main memory, known in technical terms as *random-access memory* (RAM for short). Where exactly a program is located in RAM depends on various factors. A program normally has its own address space. Therefore, an address used in a program is relative to the program's address space; it is *not* a hardware address. The need for absolute hardware addresses has largely been eliminated today.

Each object (in the meaning introduced in the Section [Initialization of Variables](#)) is located somewhere in the address space of a program. The address of its initial position in memory is a *pointer*. The same is true for functions. Every function is located somewhere in the address space of a program. A pointer to a function, i.e. a *function pointer*, is the entry point to this function.

How many bytes an address requires depends on the size of the addressed information units and their number. For example, a total of 8 gibibyte of address space is reserved for a program, and each byte of this 8 gibibyte must be addressed. One *gibibyte* (GiB) contains 1,024 *mebibytes* (MiB), one MiB contains 1,024 *kibibyte* (KiB), and one KiB holds 1,024 byte. Thus, 8,796,093,022,208 bytes must be addressed individually. The *logarithmus dualis* (abbr. *ld*) of this number gives the number of bits which are required to address all these bytes, namely $ld(8,796,093,022,208) = 43$. If the result of the calculation of *ld* would have a fractional part, it had to be rounded up to the nearest integer. Normally, addresses, object sizes, etc. are aligned with the *word size* of the computer hardware, which today is either 32 or 64 bits. Thus, a pointer to address each byte of 8 GiB has a size of 64 bits.

In C++, the *address operator*, `&`, returns the memory address of an object. To store this address, a *pointer variable*, or *pointer* for short, is required. A pointer must be defined with the type of the object to which it points, followed by an asterisk `*`. ***It is important to understand that a pointer is a variable that is supposed to contain the address of another variable. Therefore, everything that is said about variables also applies to pointers.*** The code fragment shown in Listing 21 illustrates this.

```
1 int i { 42 };  
2 int * pi { &i };
```

Listing 21: Pointer

It is worth remembering once again that in C++ the meaning of symbols can vary depending on the context in which they are used. For example, when applied to integer values, the & operator means performing a bitwise operation, and when applied to an object, it returns the object's memory address.

The first line of Listing 21 defines a variable named `i` of type `int` and initializes it to 42. The second line defines a pointer to `int` named `pi`, which is initialized to the address of variable `i` by applying the address operator, `&`, to its operand `i`.

Getting the address of an object has a complementary operation, namely dereferencing a pointer with the dereference operator, `*`. The code fragment in Listing 22 continues the code fragment of Listing 21. Therefore, the line numbers are continued.

```
3 *pi = 99;  
4 cout << i << endl;
```

Listing 22: *Dereferencing a pointer*

`*pi` dereferences the pointer. The result of the dereferencing is the object itself. This object, here the variable `i`, is assigned a new value. To see the effect of this assignment, `i` is then sent to `cout`.

Depending on the context, the asterisk, `*`, has two different meanings. In `int * pi ...`, it is part of the declaration. In `*pi = ...`, it is the *dereference operator*.

The use of a pointer can be dangerous. To illustrate this, Listing 23 shows a slightly modified code fragment.

```
1 int i { 42 };  
2 int * pi;  
3 *pi = 99;  
4 cout << i << endl;
```

Listing 23: *Dangling pointer*

The difference to Listing 21 is that in Listing 23, the pointer `pi` is not initialized when it is defined. Also, it is not assigned a valid value afterwards. Therefore, `pi` points to somewhere, perhaps to a program or hardware memory address, perhaps to a non-existent memory address. This is not a random value (as described in some articles and books), but an *undefined value*. Such a pointer is called a *dangling pointer*. When `pi` is dereferenced, there is only a small probability that it actually points to `i`. Worse, in most cases, assigning a value to `*pi` will have no noticeable effect, but it is still an error. ***An undefined pointer is a lurking danger.***

For this reason, ***a pointer should always be initialized when it is defined.*** A valid initialization value is either the address of an object or `nullptr`. `nullptr` is a keyword in C++ and denotes a value that points to nothing. Defining a pointer with zero initialization, for example `int * pi {}` has the effect of initializing the pointer

with `nullptr`. Dereferencing a pointer with value `nullptr` results in undefined behavior and may crash the program.

A pointer with value `nullptr` is converted to `false` when used in a logical expression. As the code fragment in Listing 24 shows, this can be used to dereference only valid pointers.

```
1 if (pi)
2     cout << (*pi) << endl;
3 else
4     cout << "nullptr" << endl;
```

Listing 24: Checking for valid pointer

`nullptr` can be assigned to any pointer type. As stated in Section [Fundamental Types](#), `nullptr` is of type `std::nullptr_t`. This is different for other pointer types. A pointer to `signed int` cannot be assigned to a pointer to `unsigned int`, as the code fragment in Listing 25 shows.

```
1 signed int i = 1;
2 signed int * pi = &i;
3 unsigned int j = 2;
4 unsigned int * pj = &j;
5 pj = pi; // Compile-time error: assignment between incompatible types.
```

Listing 25: Incompatible pointer types

Some older programs or libraries use `void*`, which is a pointer to `void`. A pointer of any type can be assigned to a pointer to `void`. However, the reverse is not possible. A pointer to `void` cannot be assigned to a pointer of another type without special measures. `void*` **should not be used in modern C++ programs**.

4.2.3. Stack vs. Free Store

C++ has several memory areas (*GotW #9: Memory Management - Part I*, n.d.). Two of them are the *stack* and the *free store*, which is also called *heap*. Both are of limited capacity. Normally, the stack has a smaller capacity than the heap. Each time a function is called, a new *stack frame* is pushed onto the stack. A stack frame contains, among other information, all the local variables of a function. When the function terminates, the corresponding stack frame is popped off the stack. In Section [Example for Debugging](#), Figure 13 shows an example of a function call stack, in this case the `checksum()` function of the `Checksum.cpp` program (Listing 9). When the stack is used up, attempting to push a new stack frame usually causes the program to crash.

The free store is usually larger than the stack. Therefore, a large object should be created dynamically on the free store. It is released when it is no longer needed. After that, the freed memory is available for creating other dynamic objects. A so-called *dynamic-data structure* is also created on the free store, because the memory

it requires is not known before run time. Dynamic-data structures will be explained and used later.

4.2.4. Dynamic Objects

An object created in the free store is a *dynamic object*. In the code fragment shown in Listing 26 a double variable is created as dynamic object.

```
1 double * pd { new double(42.0) };
```

Listing 26: *Dynamic object*

The new operator creates an object of the specified type, here double, on the free store. It returns the address of this object in the free store. This address is used for initializing a pointer to double called pd. The dynamic double object is created by calling its copy constructor (see Section [Initialization of Variables](#)) with the value 42.0.

Then the object can be used by dereferencing the pointer (Listing 27).

```
2 cout << (*pd) << endl; // Sends value of *pd to standard output.
```

Listing 27: *Dereferencing a pointer*

When sending the result of evaluating an expression to cout, it is advisable to enclose the expression in parentheses. In the current case, *pd would be parsed correctly. However, there are expressions, for example with bitwise << operator, that cannot be parsed correctly without parentheses.

As Listing 28 shows, a new value can also be assigned to the object. It is then sent to cout.

```
3 *pd = 99.0;
4 cout << (*pd) << endl;
```

Listing 28: *Using a dereferenced pointer*

When the object is no longer needed, it is *released* with the delete operator, as shown in Listing 29.

```
5 delete pd;
6 pd = nullptr;
```

Listing 29: *Releasing a pointer*

When an object is deleted, its *destructor* is called and the previously occupied memory is freed. A destructor is the counterpart to a constructor, which was introduced in the Section [Initialization of Variables](#). A destructor performs all the actions that must be performed when an object is destroyed. **While an object can have multiple constructors, it has exactly one destructor.** A fundamental data

type does not have an explicitly defined destructor. The destructor of a fundamental type basically does nothing.

There are several pitfalls:

- As explained earlier, a pointer which is not initialized, i.e. a dangling pointer, is invalid and should never be dereferenced.
- Also, an invalid pointer should not be deleted. If a pointer has the value `nullptr` (either by initialization or by assignment), it can be deleted without harm.
- ***A dynamic object should be deleted when it is no longer needed.*** This is not only good style, but also important to preserve available free store.
- Before assigning the address of another dynamic object to a pointer, the dynamic object the pointer actually points to should be deleted. Failure to do so will result in a *memory leak*. A program with a memory leak may run for a long time until it crashes due to insufficient free store. ***A dynamic object that no longer has a pointer pointing to it, it is completely lost. It can neither be accessed nor deleted.*** Sometimes such an object is also called an *orphan object*.
- After deleting an object, `nullptr` should be assigned to the pointer. This value indicates that this pointer should not be dereferenced. A pointer that has been deleted without `nullptr` being assigned, but is still in use, is also a dangling pointer. ***Dangling pointers are a common source of errors in C++ programs.***

Operators `new[]` and `delete[]` allow to create a variable number of dynamic objects on the free store. They are not presented here because other data types are better suited for managing a variable number of objects and are easier to use.

Even `new` and `delete` are mostly superfluous in modern C++. Other pointer types that are less error-prone and almost as efficient are introduced as needed.

4.2.5. References

A *reference* is an *alias* for an object, i.e. it is *another name for the object*. ***A reference must always be initialized in its declaration.*** There is a subtle exception that becomes relevant later: When a reference is declared as a member of a class, it can be initialized in a constructor of that class. ***A reference can only be declared, but not defined, because no memory is allocated for a reference. After it is declared, a reference cannot be changed to refer to another object. An object and a reference to it cannot be distinguished.*** They are the same object. The program shown in Listing 30 illustrates this.

¹ // References by Ulrich Eisenecker, March 18, 2021
2

```

3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     double d { 1.0 / 3.0 };
9     double & rd = d;
10    cout << "d: " << d << endl
11         << "rd: " << rd << endl;
12    rd *= 3.0;
13    cout << "d: " << d << endl
14         << "rd: " << rd << endl;
15    double & rd2 { rd };
16    cout << "rd2: " << rd2 << endl;
17    double * pd { &d },
18           * prd { &rd };
19    cout << boolalpha
20         << "pd == prd: "
21         << (pd == prd) << endl;
22 }

```

Listing 30: *References.cpp*

In line 9 of Listing 30 `rd` is declared as a reference for the variable `d` of type `double`. Here the symbol `&` is part of a declaration. Thus, it has a different meaning than the `&` operator, which takes the address of its operand, or the `&` operator, which performs a bitwise And of two integer operands. Sending `d` and `rd` to `cout` indicates that both have the same value. Changing the value of `rd` in line 12 by using the combined assignment operator, `*=`, also changes `d`. To demonstrate this, `d` and `rd` are sent to `cout` again. Since a reference cannot be distinguished from a variable, it can also be used to initialize another reference, as shown in line 15. As the ultimate test of whether a variable and a reference declared for it are indistinguishable, the addresses of `d` and `rd` are taken and the corresponding pointers are compared for equality. As the output shows, both pointers have the same value.

It is possible to declare a reference to a pointer. The program shown in Listing 31 declares a reference to a pointer to `int` named `rpi` on line 10. Thus, `pi` and `rpi` are different names for the same object.

```

1 // ReferenceToPointer by Ulrich Eisenecker, March 18, 2021
2
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     int * pi { new int { 42 } };
9     cout << "*pi: " << (*pi) << endl;
10    int *& rpi { pi };
11    if (&pi == &rpi)
12    {
13        cout << "pi and rpi have the same address, "
14             << "i.e., they are different names "
15             << "for the same object." << endl;
16    }
17    else
18    {

```

```

19     cout << "Something is broken." << endl;
20 }
21 cout << "*rpi: " << (*rpi) << endl;
22 delete pi;
23 pi = nullptr;
24 if (!rpi)
25 {
26     cout << "rpi == nullptr" << endl;
27 }
28 else
29 {
30     cout << (*rpi) << " (this should not happen)" << endl;
31 }
32 pi = new int { 99 };
33 cout << "*rpi: " << (*rpi) << endl;
34 delete pi;
35 pi = nullptr;
36 if (!rpi)
37 {
38     cout << "rpi == nullptr" << endl;
39 }
40 else
41 {
42     cout << (*rpi) << " (this should not happen)" << endl;
43 }
44 }

```

Listing 31: *ReferenceToPointer.cpp*

This is checked in line 11. Here the addresses of `pi` and `rpi` are taken with the address operator, `&`, and checked for equality. Since a pointer is also an object that resides somewhere in memory, it also has an address that is a pointer to a pointer. As the output shows, both addresses are the same. This proves that `pi` and `rpi` are different names for the same object. Therefore, it is not surprising that `rpi` points to the `int` value 42 in line 21. Deleting `pi` and assigning `nullptr` also changes `rpi`, as lines 22ff show. As soon as `pi` points to a new, dynamically allocated object, `rpi` does the same.

A reference can be used on the left side of an assignment, i.e., a new value can be assigned to it, or on the right side of an assignment, i.e., its value can be assigned to another object. An object that can appear on both sides of an assignment is called *lvalue*. An object that can appear only on the right side of an assignment is called *rvalue*. C++ allows to declare special *rvalue references* (*Reference Declaration - Cppreference.Com*, n.d.). Rvalue references are covered in detail in (Josuttis, 2020). In addition to lvalues and rvalues, there are more *value categories* in C++. See (*Value Categories - Cppreference.Com*, n.d.) for detailed information. The actual model of value categories in C++ is more advanced than the simple distinction between lvalues and rvalues. However, for this text this simplified distinction is sufficient.

4.2.6. Constants

Sometimes a variable is initialized once and it should not change its value after that. For example, the variable `e` of type `long double` is initialized with a value close to *Euler's number* (Listing 32).

```
1 long double e { 2.71828'18284'59045'23536'02874'71352'66249'77572'47093L }
```

Listing 32: *Euler's number*

Without further precautions, the value of `e` can be changed anywhere in the program after `e` has been initialized. To prevent this, `const` is prepended to the definition of `e` (Listing 33).

```
1 const long double e { 2.71828'18284'59045'23536'02874'71352'66249'77572'47093L }
```

Listing 33: *Euler's number as constant*

Now the compiler refuses any attempt to change the value of `e`.

`const` can be combined with references and with pointers. Since `e` is now `const`, it is not possible to declare an ordinary reference to it. For example, `long double & Euler { e }`; is rejected by the compiler because the value of `e` could be changed by changing `Euler`. But it is possible to declare a reference to `const` (Listing 34).

```
1 const long double & Euler { e };
```

Listing 34: *Reference to const for const object*

Since this reference retains the *constness* of `e`, its declaration is perfectly legal. It is also possible to declare a reference to `const` for a non-`const` object (Listing 35).

```
1 char myChar { 'u' };
2 const char & notReallyConst { myChar };
3 // notReallyConst = 'x'; // Illegal, notReallyConst is a reference to const
4 myChar = 'x'; // perfectly legal
5 cout << notReallyConst << endl; // Value of notReallyConst changed via myChar.
```

Listing 35: *Reference to const for non-const object*

Reference to const and *const reference* should not be confused (“The Incredible Const Reference That Isn’t Const,” 2018). After a reference is initialized, it cannot be changed to refer to another object. It does not matter whether it is an ordinary reference or a reference to a `const` object. As explained earlier, it is even perfect to have a reference to `const` for a non-`const` object. But a `const` reference is actually a reference declared as `const`. With some effort it is possible to declare such a `const` reference, but hopefully the compiler will warn that a `const` qualifier on a reference type has no effect.

A pointer can be combined with `const` in several ways:

1. Define a pointer to a `const` object.
2. Define a `const` pointer to an object.

3. Define a const pointer to a const object.

The program shown in Listing 36 illustrates all combinations.

```
1 // PointerAndConst by Ulrich Eisenecker, March 18, 2021
2
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     int * pi { new int { 42 } };
9     cout << "*pi: " << (*pi) << endl;
10
11     // pointer to const
12     const int * pci1 { pi }; // pointer to const int
13     int const * pci2 { pi }; // also pointer to const int
14     cout << "*pci1: " << (*pci1) << endl; // Ok, using *pci1 as rvalue
15     // (*pci2) = 99; // Error, pointer to const cannot be modified
16
17     // const pointer
18     int * const cpi { new int { 11 } }; // const pointer to int
19     cout << "*cpi: " << (*cpi) << endl;
20     (*cpi) = 99; // Ok, const pointer points to non-const int
21     cout << "*cpi: " << (*cpi) << endl;
22     delete cpi; // Ok, const pointer may be deleted!
23     // cpi = nullptr; // Error, const pointer cannot be modified
24     // --> invalid pointer
25
26     // const pointer to const
27     const int * const cpci1 { pi }; // const pointer to const int
28     int const * const cpci2 { pi }; // also const pointer to const int
29     // (*cpci1) = 99; // Error, const int cannot be modified
30     // cpci1 = cpci2; // Error, const pointer cannot be modified
31     // delete cpci1; // Bad idea! Dynamic object is owned by pi, not by cpci1
32     // cpci1 = nullptr; // Error, const pointer cannot be modified
33     // --> invalid pointer
34     // cpci2 = nullptr; // Error, const pointer cannot be modified
35     // --> invalid pointer
36
37     delete pi; // Ok, *pi is owned by pi, thus it should be released by pi
38     pi = nullptr; // Ok
39 }
```

Listing 36: *PointerAndConst.cpp*

In line 8 `pi` is defined as a pointer to `int` and initialized with the address of a newly created `int` object, which is initialized to 42. In a sense, `pi` is the owner of this dynamic object because it was initialized with the address of this object. Because of this implicit ownership, the dynamic object is released by `delete pi`; at line 37. Then `pi` is assigned `nullptr` to indicate that it should not be dereferenced.

4.2.6.1. Pointer to const

Lines 11 through 15 in Listing 36 show pointers to const objects. `const` must come either immediately before or after the type identifier. Therefore, `const int` and `int const` are equivalent. `const int` is used more frequently. Next to the right, the asterisk, `*`, indicates that a pointer is declared.

As shown in line 14, a dereferenced pointer can be used as rvalue, i.e. its value can be accessed, but it must not be changed. It must not appear as lvalue (line 15), i.e. it cannot be assigned a new value, as this would imply changing the const object pointed to by the pointer. After commenting out line 15, the compiler should generate an appropriate error message.

4.2.6.2. const Pointer

Line 18 of Listing 36 declares a const pointer to int. If const were placed before the asterisk, *, it would belong to the type. Therefore, the only valid place for const to declare a const pointer is directly after the asterisk.

Since the pointer is const, it must be initialized when it is defined. Here it is initialized with the address of a dynamically created int object with the value 11. This object can be used as both an rvalue (line 19) and an lvalue (line 20). Line 22 illustrates something important. A const pointer can point to an object that was created dynamically, as is the case in line 18. Therefore, it must be possible to apply the delete operator to a const pointer to release the dynamically created object (line 22). This returns the memory space of the dynamic object to the free store. However, the value of the pointer is not changed. As line 23 shows, it is not possible to assign nullptr to the const pointer cpi. Therefore, the recommendation given earlier cannot be followed. This makes the const pointer very dangerous, since now it remains invalid without any possibility to change it.

4.2.6.3. const Pointer to const

Lines 27 and 28 of Listing 36 illustrate two ways to declare a const pointer to a const object. Consequently, both ways are a combination of declaring a pointer to const and declaring a const pointer.

A possible error is displayed in line 31. In line 27 cpci1 was initialized with the value of pi. For this reason, cpci1 does not own the object to which pi points. Nevertheless, it is possible to apply the operator delete to cpci1, which releases the object owned by pi. Since pi is unaware of this, there is a risk of trying to release the object a second time by applying the delete operator to cpci1. This would lead to undefined behavior. As with cpi, cpci1 cannot be assigned nullptr. After delete cpci1;, cpci1 remains an invalid pointer until the end of its lifetime.

4.2.6.4. Naming

In the PointerAndConst.cpp program (Listing 36), the identifiers for variables were created systematically. pi can be thought as an acronym of *pointer (to) int*. Thus, the

name `pi` reflects the declaration in reverse order. `pci2` is an acronym of *pointer (to) const int* with an attached cipher. `cpci2` can be thought of as an acronym for *const pointer (to) const int*. The characters of the two identifiers reverse the order of the corresponding parts of their declarations. In the connection with the explanation of complex declarations these identifiers can be regarded as didactically helpful.

The best-known approach to encoding information about type or intended use in variable names is the *Hungarian notation* invented by Charles Simonyi (“Hungarian Notation,” 2021). It was once very popular and has advantages for programming languages that are not strongly typed. When using modern strongly typed programming languages, this naming convention is rather unadvisable. First, types would be represented twice, as the type itself and in the name of a variable of that type. Second, in object-oriented programming, the dynamic type of an object may vary, and generic programming supports writing code independent of concrete types. In these cases, it would be pointless to encode type information in variable identifiers.

4.3. More About Functions

Functions have already been mentioned several times, e.g., in the Sections `main()`, `Function – First Information`, `Unit Tests` and `Example for Debugging`. Although functions are not mandatory for Turing-completeness, they are a very important concept of programming languages. Their most important purpose is to assist human programmers by satisfying the following criteria:

1. The implementation of a function should be of limited size. Perhaps a maximum length of 24 lines, none of which exceeds 80 columns, is a well-chosen upper limit. Longer implementations take too much time to read and to understand. If a function implementation is longer, it should be split into several functions.
2. A function should be well named. Its name should be concise and express the intention of the function. Anything else is misleading. Like a variable identifier, the name of a function helps the human programmer to read, understand, remember and recognize the code.
3. **A function should serve a single purpose**, i.e., a) it either performs a computation, b) it takes an input, c) it produces an output, or d) it implements an interaction, i.e., a sequence of related inputs and outputs, with an actor, e.g., a human user or another program. In doing so, it may call any necessary functions which may fall into any of the four categories. A function should not serve more than one of these purposes. It is acceptable for a function to additionally check whether it can perform its task without error or to correct

error conditions reported to it, e.g., by a caught exception. However, this should not excessively increase the size of the function.

Like the `Checksum.cpp` program (Listing 9), the program in Listing 37 calculates the checksum of a natural number. But there are some differences:

1. The variables `number` and `checksum` are of type `unsigned int`, not `unsigned long int`. This is only a minor difference.
2. Calculating the checksum is part of `main()`, there is no separate function for it.

Here, the checksum is calculated iteratively using a `while` loop. The `checksum()` function in the `Checksum.cpp` program has a recursive implementation instead.

The `main()` function in the `ChecksumMain.cpp` program (Listing 37) has only 18 lines. This is a moderate size that does not in itself require subdivision into further functions.

```
1 // ChecksumMain by Ulrich Eisenecker, November 24, 2023
2
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     cout << "Natural number: " << flush;
9     unsigned int number {};
10    cin >> number;
11    unsigned int checksum {};
12    while (number > 0)
13    {
14        checksum = checksum + number % 10;
15        number = number / 10;
16    }
17    cout << "Checksum = " << checksum << endl;
18 }
```

Listing 37: `ChecksumMain.cpp`

The name `main()` says nothing about the intention of the function, apart from the fact that it serves as an entry point for starting the program. It certainly does not serve only one purpose. It reads in a value, it calculates a value, and it outputs the calculated value. Reading the input and generating the output belong together and form an interaction. The user provides an input and receives a corresponding output. `main()` thus has two purposes, performing a calculation and perform an interaction. Combination with the shallow name, this is a strong argument for introducing an additional function.

Therefore, this program is rewritten as shown in Listing 38. In the new version, a separate function for calculating the checksum is introduced. Both programs should be identical in terms of their functionality and external behavior. ***The process of restructuring a program or a part of a program is called refactoring.***

```
1 // ChecksumFunction by Ulrich Eisenecker, November 24 2023
```

```

2
3 #include <iostream>
4 using namespace std;
5
6 unsigned int checksum(unsigned int number)
7 {
8     unsigned int result { };
9     while (number > 0)
10    {
11        result = result + number % 10;
12        number = number / 10;
13    }
14    return result;
15 }
16
17 int main()
18 {
19     cout << "Natural number: " << flush;
20     unsigned int number { };
21     cin >> number;
22     cout << "Checksum = " << checksum(number) << endl;
23 }

```

Listing 38: *ChecksumFunction.cpp*

The program in Listing 38 has two functions and 23 lines of source code, or shorter, *lines of code* (abbr. *LoC*). Compared to its predecessor, the number of functions has doubled, and the LoC is increased by 27.8 %. A larger program takes more time to read and can be more difficult to understand. In fact, the original version has a ratio of 18 LoC per function, while the refactored version has a ratio of only 11.5 LoC per function. This decrease may indicate that each function is actually easier to read and understand.

The `checksum()` function takes one parameter, namely `number` for which the checksum is to be calculated. It defines the variable `result` of type `unsigned int` and initializes it to 0. This variable is needed for the iterative calculation of the checksum. Its name is appropriate, because it actually contains only intermediate results and finally the final result that `checksum()` returns to the caller. This function is short, has a meaningful name and a single purpose.

The `main()` function of the revised program is considerably shorter compared to the `main()` function of its predecessor. All the code for calculating the checksum has been moved to the `checksum()` function. Only the interaction with the user is still present in `main()`, which is a clear single purpose.

Also, `main()` is now easier to understand. In the previous version, it is difficult to figure out what the calculations do. By calling the `checksum()` function, the purpose and implementation of `main()` are now easier to understand.

Overall, the refactoring can be considered successful. All these improvements are only relevant for the human programmer. The refactoring does not bring any advantage for the compiler.

The `checksum()` function illustrates another purpose for creating functions. Whenever a set of instructions corresponds to a mathematical function, it is worth thinking about implementing it as a function. Therefore, a mathematical function should have its counterpart as a function in a program.

The only effect of the `checksum()` function is that it returns a value. This value must therefore be used in some way for the call of the function to have any effect at all. It is a legacy of *C*, the predecessor of *C++*, that the return value of a function can simply be ignored, e.g., `checksum(42);`. To prevent this, it is possible to declare the function with the `[[nodiscard]]` attribute, as shown in Listing 39.

```
1 [[nodiscard]] unsigned int checksum(unsigned int number)
2 {
3 // ...
```

Listing 39: `checksum()` with `[[nodiscard]]` attribute

Now the compiler issues a warning if the return value is not used at all. ***It is recommended to use `[[nodiscard]]` whenever the only effect of a function is to return a value.***

4.3.1. Declaration vs. Definition

In *C++* it is important to distinguish between *declaration* and *definition*. In general, ***a declaration introduces a name and a meaning***, so that the entity is known to the compiler. ***A definition introduces additional information so that the entity can also be used in any way***, e.g. required memory for data or code or the structure of a data type. In the case of a function, the declaration corresponds to the so-called *function prototype*, and the definition additionally includes the function implementation.

In the code snippet of Listing 40, the declaration that is italicized is also a definition of the `isEven()` function.

```
1 bool isEven(int number)
2 {
3     if (number % 2 == 0)
4     {
5         return true;
6     }
7     else
8     {
9         return false;
10    }
11 }
12
13 int main()
14 {
15     // ...
16 }
```

Listing 40: Declaration, which is also a definition

This is because there is no preceding declaration without a definition. Therefore, this declaration is both a declaration and a definition. After this declaration, which is also a definition, the function `isEven()` can be called in function `main()`.

The code snippet shown in Listing 41 contains a declaration of the `isEven()` function, which is in bold, and its definition, which is in italics.

```
1 bool isEven(int number);
2
3 int main()
4 {
5     // ...
6 }
7
8 bool isEven(int number)
9 {
10     if (number % 2 == 0)
11     {
12         return true;
13     }
14     else
15     {
16         return false;
17     }
18 }
```

Listing 41: Declaration and separate definition

After the declaration without definition the `isEven()` function can be also used in the `main()` function. But without the definition that follows `main()`, the program cannot be linked because the implementation of `isEven()` is missing. In fact, the definition of `isEven()` could be placed somewhere between the declaration of `isEven()` and `main()`, after the implementation of `main()` (as shown here), or it could be part of another source file that had to be compiled and linked along with the object file containing `main()`.

It would be syntactically legal to omit the name of the parameter in the function declaration, i.e., `bool isEven(int);`. However, this should never be done because the names of the parameters in the function declaration may have meaning for other programmers.

In C++, the so-called One Definition Rule (abbr. ODR) states that an entity may have several (identical) declarations, but only one definition. A violation of the ODR, e.g. by providing two differing definitions for an entity or by repeating a definition in the same translation unit, leads to a compilation error.

4.3.2. Passing Parameters and Returning Results

A function declaration can contain zero or more parameters. For each parameter it is necessary to specify how it will be passed when the function is called. It is possible to provide default values for parameters that will be used when the

function is called with fewer parameters. These default values can be omitted from right to left. It is also possible to declare functions that accept any number of parameters. In addition, there can be multiple functions with the same name, as long as they differ in the number or types of their parameters. This is called *overloading* and will be also discussed. When declaring functions, the programmer must be aware of possible conflicts that may arise from combining the above possibilities when declaring functions.

There are also several ways to return results of function calls, which will also be explained.

4.3.2.1. Call by Value

All functions presented so far use *call-by-value* as the mechanism for passing parameters. When a parameter is passed by value, it is copied before entering the function. In the function itself, the copy is accessed by the name specified in the declaration. Within the function this copy can also be changed, but this only affects the current execution of the function. However, this has no effect on the actual parameter with which the function was called, since only its copy is affected by this change.

The program in Listing 42 illustrates the aforementioned details.

```
1 // CallByValue by Ulrich Eisenecker, April 1, 2021
2
3 #include <iostream>
4 using namespace std;
5
6 void someFunction(double number)
7 {
8     cout << "Entering someFunction() ... \n"
9     << "Memory address of number = "
10     << (&number)
11     << endl
12     << "Value of number = "
13     << number
14     << endl;
15     number = number * 2.0;
16     cout << "New value of number = "
17     << number
18     << "\n... leaving someFunction()"
19     << endl;
20 }
21
22 int main()
23 {
24     double number { 21.0 };
25     cout << "In main() ... \n"
26     << "Memory address of number = "
27     << (&number)
28     << endl
29     << "Value of number = "
30     << number
31     << endl;
32
```

```

33     someFunction(number);
34
35     cout << "... back in main()\n"
36     << "Memory address of number = "
37         << (&number)
38     << endl
39     << "Value of number = "
40     << number
41     << endl;
42 }

```

Listing 42: *CallByValue.cpp*

The function `someFunction()` has only one parameter named `number`, which is of type `double` and passed by value. First, `someFunction()` outputs the memory address of `number`. This is the address of the local variable `number`. Then the current value of `number` is sent to `cout`. Next, the local variable `number` is changed. This change is effective, as shown by sending `number` to `cout` again.

In `main()` `number` is declared as `double` and initialized with `21.0`. Then its memory address and its value are sent to `cout`. Then `someFunction()` is called with `number` as parameter. Finally, the memory address and the value of `number` are sent to `cout` again.

Figure 20 shows a sample dialog for running this program, assuming that the program has been compiled into an executable called *CallByValue*.

```

./CallByValue
In main() ...
Memory address of number = 0x7ffeec17aaf8
Value of number = 21
Entering someFunction() ...
Memory address of number = 0x7ffeec17aac8
Value of number = 21
New value of number = 42
... leaving someFunction()
... back in main()
Memory address of number = 0x7ffeec17aaf8
Value of number = 21

```

Figure 20: *Sample dialog for executing the CallByValue.cpp program*

In a *command line window* (also called *terminal* or *shell*) of Unix-like operating systems, the name of an executable binary program must be preceded by the path where it is located. The dot stands for the active directory and the slash separates the directory name from the name of the program, which here is simply *CallByValue* (without extension).

Executing this program a second time or on another platform, the addresses of `number` in `main()` and `number` in `someFunction()` may differ. The important observation is that `number` in `someFunction()` and `number` in `main()` have different addresses. This is because `number` in `someFunction()` is a new variable which is initialized with the value of the passed parameter. The local variable in

`someFunction()` can be changed, and this change is subsequently effective, but only as long as the current execution of `someFunction()` is active. As soon as `someFunction()` terminates, the local variable `number` of `someFunction()` is destroyed. `number` in `main()` is not affected by the change of its local copy in `someFunction()`.

The primary effect of passing a parameter by value is that the original parameter cannot be changed within the function. The ability to change the value of the copied parameter within the function is usually not of interest and may even be undesired. To prevent the copied parameter from being changed, it can be passed as a `const` parameter. Accordingly, `someFunction()` has to be defined as shown in Listing 43.

```
1 void someFunction(const double number)
2 // ...
```

Listing 43: *Pass call-by-value parameter as const*

Now the compiler reports an error at line 15 of Listing 42, where `number` is assigned a new value.

Call by value is the only parameter passing mechanism available in *C*. *C++* provides additional mechanisms for passing parameters for various purposes. Some authors, e.g., (Breymann, 2023) and (Gregoire, 2020), recommend passing parameters of fundamental types by value in general, additionally declared as `const`, if they must not be modified locally. They argue that this is the most efficient parameter passing mechanism for data types that have fewer or the same number of bytes as the CPU's word width, which is typically 64 bits, because the CPU processes a word in a single step. But there is no advantage of passing a byte over passing a `double` by value, and larger data types incur more overhead. The alternative, passing a parameter as a reference to `const`, presented later, incurs a constant overhead for passing all types of parameters that is comparable to the call by value. Therefore, passing a parameter by value or as a `const` parameter by value is discouraged in this text. Exceptions are parameters of recursive functions that must be passed by value in order to allow recursion, or cases where changing the local copy of a parameter allows a shorter implementation.

4.3.2.2. Call by Pointer

In *C*, call by value is the only mechanism for passing parameters. Therefore, passing a large data type to a function takes time to create the copy, and the copy consumes memory. In addition, it is not possible to change the parameter passed by value. By passing a *pointer by value*, both problems can be circumvented.

The program in Listing 44 illustrates passing a pointer by value.

```
1 // CallByPointer by Ulrich Eisenecker, April 6, 2021
2
```

```

3 #include <iostream>
4 #include <string> // because of string
5 using namespace std;
6
7 void prependSalutation(string * name)
8 {
9     cout << "Entering prependSalutation() ... \n"
10         << "Memory address of name = "
11         << (&name)
12         << endl
13         << "Content of name = "
14         << name // This is a memory address!
15         << endl
16         << "Value of *name = "
17         << (*name)
18         << endl;
19     *name = "Hi, " + *name;
20     cout << "New value of *name = "
21         << (*name)
22         << "\n... leaving prependSalutation()"
23         << endl;
24 }
25
26 int main()
27 {
28     string yourName { };
29     cout << "In main() ... \n"
30         << endl
31         << "Your name: "
32         << flush;
33     cin >> yourName;
34     cout << "Memory address of yourName = "
35         << (&yourName)
36         << endl
37         << "Value of yourName = "
38         << yourName
39         << endl;
40
41     prependSalutation(&yourName);
42
43     cout << "... back in main() \n"
44         << "Memory address of yourName = "
45         << (&yourName)
46         << endl
47         << "Value of yourName = "
48         << yourName
49         << endl;
50 }

```

Listing 44: *CallByPointer.cpp*

The `prependSalutation()` function prepends "Hi, " to the string pointed to by the `name` parameter. `name` is of type `std::string`, which is the type for representing and manipulating strings that is part of the C++ standard library. To use it, `<string>` must be included. The parameter specification `string * name` states that a pointer to string named `name` is passed by value. For this reason, line 41 of Listing 44 uses the address operator, `&`, to take the address of the string variable `yourName` defined in `main()`, namely `prependSalutation(&yourName);`. Figure 21 shows an example dialog for executing the program compiled to an executable binary named *CallByPointer*.

```

./CallByPointer
In main() ...

Your name: Harley
Memory address of yourName = 0x7ffefeea86a68
Value of yourName = Harley
Entering prependSalutation() ...
Memory address of name = 0x7ffefeea869c8
Content of name = 0x7ffefeea86a68
Value of *name = Harley
New value of *name = Hi, Harley
... leaving prependSalutation()
... back in main()
Memory address of yourName = 0x7ffefeea86a68
Value of yourName = Hi, Harley

```

Figure 21: *Sample dialog for executing the CallByPointer.cpp program*

Again, the memory addresses may be different if the program is run another time or on another platform. It is important that the memory address of `yourName`, which is passed to `prependSalutation()` in line 41 of Listing 44 and the memory address of `name` in `prependSalutation()` are different! `&yourName` is the memory address where the string `yourName` starts. This address is passed by value to `prependSalutation()`. Taking the address of `name` by `&name` gives the memory address where the pointer of the copy of `&yourName` is located. Thus `&name` is a pointer to a pointer to string. Apart from the fact that pointer to string and pointer to pointer to string are different types, both addresses should be the same if the pointer `name` had not been copied when passed to `prependSalutation()`. It may be more illustrative to look at the content that `name` points to, namely `*name`. This is exactly the address of `yourName` in the `main()` function! So the statement `*name = "Hi, " + *name;` on line 19 of Listing 44 actually modifies the string `yourName` defined in `main()`. It should be mentioned that the `+` operator allows a `std::string` to be concatenated with another string, be it a C string (as in this case), or a C++ string. Figure 22 illustrates this scene with pointers.

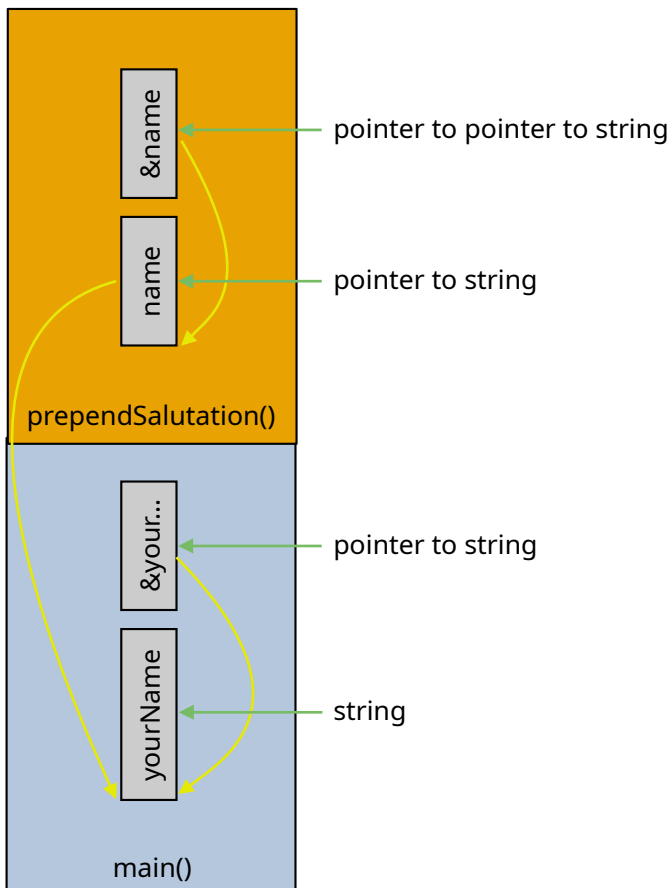


Figure 22: Variety of pointers

In the program `CallByValue.cpp` (Listing 42) the value of `number` can be changed in the `someFunction()` function. The same is true for `name` in `prependSalutation()` (Listing 44). It is possible to assign a new value to `name`. This must be a pointer to string, e.g. `name = name;` (which is the most harmless variant of assigning a new value in this case) which is inserted as the first line in `prependSalutation()`. To prevent this, a const pointer must be passed. Therefore, the `const` keyword must be placed to the right of the asterisk, namely `void prependSalutation(string * const name)`. After this change, the assignment `name = name;` can no longer be compiled.

Suppose the only purpose of passing a pointer as a parameter would be to save time and memory for making a copy, but the memory pointed to by the pointer is not to be changed. In this case, a pointer to a const parameter must be specified. Unfortunately, this can be done in two equivalent ways:

1. `void prependSalutation(const string * name)` and
2. `void prependSalutation(string const * name)`.

That is, `const` must be placed either to the left of the type or to the right of it. In either case, attempting to modify `*name` as in line 19 of Listing 44 is no longer compilable. Besides, `void prependSalutation(const string const * name)` is

syntactically illegal, and the compiler would report an error. Of course, it is possible to pass a const pointer to a const parameter, for example `void prependSalutation(const string * const name)`, or `void prependSalutation(string const * const name)`. In both cases, neither the pointer itself nor the memory it points to can be modified.

A rather questionable purpose of passing a pointer as a parameter is to change it by a function. There must be a good reason for this, which is not the case in the `prependSalutation()` function. It is better to let the caller of `prependSalutation()` decide whether to modify `yourName`, define a new variable and initialize it with the result of the function call, send the result to `cout` without using it further, or to ignore the result of the function call altogether. Listing 45 shows a corresponding function declaration that allows all this.

```
1 string prependSalutation(string * const name)
2 {
3     return "Hi, " + *name;
4 }
```

Listing 45: *const pointer as function parameter*

4.3.2.3. Call by Reference

Another mechanism for passing a parameter to a function in C++ is *by reference*. Consider a function that takes two natural numbers and calculates the result of integer division and its remainder. One way to get both results is to pass all parameters by reference, as the program in Listing 46 shows.

```
1 // CallByReference by Ulrich Eisenecker, April 6, 2021
2
3 #include <iostream>
4 using namespace std;
5
6 void integralDivision(unsigned int & dividend, unsigned int & divisor,
7                     unsigned int & quotient, unsigned int & remainder)
8 {
9     cout << "Entering integralDivision() ... \n"
10         << "Memory address of dividend = "
11         << (&dividend)
12         << endl;
13     quotient = dividend / divisor;
14     remainder = dividend % divisor;
15 }
16
17 int main()
18 {
19     unsigned int dividend { 99 },
20               divisor { 16 },
21               quotient { },
22               remainder { };
23     cout << "In main() ... \n"
24         << "Memory address of dividend = "
25         << (&dividend)
26         << endl;
27
28     integralDivision(dividend, divisor, quotient, remainder);
```

```

29
30     cout << "... back in main()\n"
31         << dividend
32         << " integrally divided by "
33         << divisor
34         << " is "
35         << quotient
36         << ", remainder "
37         << remainder
38         << '.'
39         << endl;
40 }

```

Listing 46: *CallByReference.cpp*

A reference to a parameter is simply declared by placing an & after the type name and before the parameter name. The `integralDivision()` function has four reference parameters. The first two serve as *input parameters*, namely `dividend` and `divisor`. The second two serve as *output parameters*, namely `quotient` and `remainder`. After the function call is completed they contain the corresponding results.

After compiling the program to an executable file named *CallByReference*, the dialog shown in Figure 23 appears.

```

./CallByReference
In main() ...
Memory address of dividend = 0x7ffee810ca7c
Entering integralDivision() ...
Memory address of dividend = 0x7ffee810ca7c
... back in main()
99 integrally divided by 16 is 6, remainder 3.

```

Figure 23: *Output of the CallByReference.cpp program (Listing 46)*

As an example, the addresses of the `dividend` variable in `main()` and of the `dividend` parameter of `integralDivision()` are sent to `cout`. As the dialog shows, both addresses are identical. That is, `dividend` in `integralDivision()` and `dividend` in `main()` are the same object! Here too, the addresses can change if the program is executed again or on a different platform.

The same is true for all other variables defined in `main()` that are also passed by reference to `integralDivision()`. This effect is only related to the fact that these parameters are passed by reference, and not to the idiosyncratic fact that the same names are used for the variables of `main()` and the parameters of `integralDivision()`.

Two problems remain. First, the `integralDivision()` function cannot be called with literals as the first two parameters, for example, `integralDivision(98u, 17u, quotient, remainder)`. Second, within the `integralDivision()` function the first two parameters could be changed, although the caller would probably not like this, for example `divisor = 0;`. Both problems can be easily fixed by passing `dividend`

and divisor as *reference-to-const parameters*. Listing 47 shows the correspondingly updated function prototype.

```
1 void integralDivision(const unsigned int & dividend, const unsigned int & divisor,  
2                       unsigned int & quotient, unsigned int & remainder);
```

Listing 47: *integralDivision()* function with reference-to-const parameters

Again, it would also be syntactically valid to place `const` after the type name as Listing 48 shows.

```
1 void integralDivision(unsigned int const & dividend, unsigned int const & divisor,  
2                       unsigned int & quotient, unsigned int & remainder);
```

Listing 48: *Alternative declaration of integralDivision()* function

After this change it is possible to call `integralDivision(98u ,17u ,quotient, remainder)`, where `dividend` and `divisor` must not be changed in `integralDivision()`.

When passing `dividend` and `divisor` as reference-to-const parameters, two objects are in the stack frame of the corresponding function, i.e. they have valid addresses in the memory. However, literals neither exist in the same memory area as normal variables nor can their addresses be taken. Rather, for a literal that is used as parameter, the compiler automatically creates a temporary object of the desired type that is subsequently passed as a parameter. Since this temporary object is constant, the compiler assumes that this object is used read-only and can be safely destroyed when the function is finished.

For this reason, the following general advice is given:

1. ***A function parameter should be passed by reference to `const` if its value is not to be changed by the function.***
2. ***A function parameter should be passed by reference, if it is not possible to return it as a result of the function.***
3. ***A function parameter must be passed as a value if a) its direct modification allows a more compact implementation of the function, or if b) there must be individual instances of this parameter per function call due to a recursive implementation of the function.***

In addition, the C++ compiler can perform a significant optimization when a parameter is passed by value, which is not possible when it is passed by reference to `const`. However, explaining this optimization and describing its requirements is beyond the scope of this text.

Aside it is possible to pass a pointer by reference, so that a change of the pointer within the function affects the pointer of the calling site, for example `void someFunction(int * & ip);`. This is only mentioned for completeness. ***The use of pointers should be avoided.***

4.3.3. Returning Results

All mechanisms presented for passing parameters to functions also apply to returning function results. ***While a function can take any number of parameters, it returns none or exactly one result.*** The way in which a result is returned is mostly a matter of design and depends on the scope and lifetime of the result. The details are discussed below.

An example is introduced for illustrative purposes. One transformation that can be applied to a `std::string` (in the following just `string` for brevity) is to reverse the order of its characters, i.e. *Hello* becomes *olleH*.

4.3.3.1. Return Type `void`

The first variant of such a function assumes that it accepts only one reference parameter, *ref-parameter* for short, namely a `string` to be reversed, and the `string` is reversed in itself. Thus, the function returns nothing, that is, its return type is `void`. The function prototype that reflects the above assumptions is `void reverseString1(string& s);`.

There will be several versions and variants of this function. Therefore, a number is appended to its name to distinguish them. Before implementing this function, some more information about the type `string` is required and the algorithm for reversing the order of characters must be carefully planned. It should be mentioned that the C++ standard library already provides the algorithm function template `std::reverse<>()` for reversing the elements of a container such as a `string`. In professional programming one would normally refrain from implementing an additional version.

1. The type `string` has various so-called *member functions* and *member types*. The member type `string::size_type` specifies the type used to represent information about the length of a `string` or to index its elements. A for loop that explicitly iterates over the individual characters of a `string` should use a loop variable of type `string::size_type`.
2. The member function `string::length()` returns the actual length of a `string`, i.e. the number of its characters, as `string::size_type`. The member function `string::size()` behaves exactly the same. Another alternative is the free function template `std::size<>()`, which calls `std::string::size()` and returns the result of the function call.
3. The member function `string::at(string::size_type pos)` returns the character at position `pos` as a reference to type `char`. The characters of a `string` `s` are indexed from 0 to `s.length() - 1`. If `pos` is outside this range, `string::at()` throws a so-called *exception*, which causes a run-time error. An alternative to `string::at()` is the index operator `string::oper-`

ator[string::size_type pos], which does *not* perform a range check. If pos is out of range, the program behavior is undefined.

4. It is possible to specify string literals in a program. This is done by appending the suffix `s` to the string literal, e.g. `"This is a C++-string."`s. The string suffix operators – there are others for related string types – are declared in the namespace `std::literals::string_literals`, which can also be accessed through the namespace `std::literals` or `std::string_literals`.

For the planning of the algorithm it is advantageous to consider the details of the data to be manipulated. Table 10 shows the individual characters of the C++ string "Hello"s in the last row, their natural indices in the first row and their C++ indices in the middle row. The natural indices start with 1, the C++ indices with 0. Regardless of the different indices, the length of "Hello"s is 5.

Natural index	1	2	3	4	5
C++ index	0	1	2	3	4
Character	'H'	'e'	'l'	'l'	'o'

Table 10: C++ string "Hello"s

First, one can imagine reversing a string with natural indices. In this way, the following positions need to be swapped:

1. 1, 5
2. 2, 4
3. 3, 3

Performing additional permutations, namely 4, 3 and 5, 1, would undo the permutations already done! Also the swapping of 3 and 3 is superfluous, because it has no effect. Now 5, which refers to the last character of the string, is replaced by `"Hello"s.length()`, or more generally, by `s.length()`, which will be abbreviated to l (short for *length*) in the following.

Still using natural indices, it is now clear that swapping must stop at the position $l/2$, whereby $/$ denotes the integer division. The result of an integer division is an integral value without a decimal places. This gives the last valid value for the 1st index for swapping, which is called m (short for *middle*). To calculate the 2nd index to swap, a (short for *alternate index*), the current index, let's call it i (short for *index*), must be subtracted from l .

Since the initial value of i is 1, this results in index positions being systematically too small by 1. Therefore, 1 must be added, which results in $l - i + 1$. Table 11 summarizes the algorithmic quantities just introduced.

Table 12 shows all pairs of values i , a , i.e., 1st index and 2nd index, calculated according to Table 11 for values i from 1 to m for "Hello"s, resulting in natural indices.

Symbol	Formula	Meaning
l	<code>s.length()</code>	Length of string
m	$l / 2$	l integrally divided by 2
i	[1.. m]	1 st index for swapping; initial value is 1, last value is m
a	$l - i + 1$	2 nd index for swapping

Table 11: Algorithmic quantities for reverting a string (natural index values)

Iteration	i (1 st index)	a (2 nd index)
1	1	5
2	2	4

Table 12: Natural index values for reverting "Hello"s

The index positions of Table 12 are calculated according to the natural index starting with 1. To obtain C++ indices, 1 must be subtracted from each natural index value. A viable alternative is to adjust Table 11 for C++ index values, which is done in Table 13.

Symbol	Formula	Meaning
l	<code>s.length()</code>	Length of string
m	$l / 2 - 1$	l integrally divided by 2, minus 1 to obtain C++ index
i	[0.. m]	1 st index for swapping; initial value is 0, last value is m
a	$l - i - 1$	2 nd index for swapping

Table 13: Algorithmic quantities for reverting a string (C++ index values)

Table 14 shows the value pairs i , a calculated as C++ indices. Additionally, the corresponding characters of "Hello"s are shown.

Iteration	i	a	<code>s.at(i)</code>	<code>s.at(a)</code>
1	0	4	'H'	'o'
2	1	3	'e'	'l'

Table 14: C++ index values for reverting "Hello"s

This way of developing an algorithm for reversing a string may seem tedious and complicated. Nevertheless, it is thorough and appropriate. First, the problem is

analyzed using the concrete string "Hello"s, which consists of individual characters referenced by natural indices starting at 1. Second, the problem is generalized to strings of arbitrary length and formulated in terms of symbols with associated formulas. Third, the formulas are adapted with respect to C++ indices starting at 0. ***These steps form a fundamental pattern of problem solving, namely***

- 1. Analyzing a problem using a concrete example.***
- 2. Creating an abstract description of the problem and working out a solution to the abstract problem.***
- 3. Refining the abstract solution to fit the actual circumstances.***

The program in Listing 49 shows a congruent implementation of this algorithm in the reverseString1() function and a main() function for testing.

```
1 // ReturnVoid by Ulrich Eisenecker, April 8, 2021
2
3 #include <iostream>
4 #include <string> // because of string
5 #include <algorithm> // because of swap()
6 using namespace std;
7
8 void reverseString1(string & s)
9 {
10     for (string::size_type i { 0 },
11         l { s.length() },
12         m { l / 2 - 1 };
13         i <= m; ++i)
14     {
15         swap(s.at(i),s.at(l - i - 1));
16     }
17 }
18
19 int main()
20 {
21     string someString { };
22     cout << "String: "
23         << flush;
24     cin >> someString;
25     reverseString1(someString);
26
27     cout << "Reversed string = "
28         << someString
29         << endl;
30 }
```

Listing 49: ReturnVoid.cpp

The implementation of reverseString1() contains exactly one statement, a for loop. The header of this for loop consists of three parts separated by semicolons. The first part initializes the variables i, l, and m. These variables are only visible inside the for loop. As described above, string::size_type is the type of all size- or index-related members of string. i is explicitly initialized to 0 by { 0 }. Simply { } would have had the same effect, but for the reader { 0 } may be more concise. l is initialized to s.length() because it is needed multiple times. Therefore, it is advisable to cache the result of s.length() in a variable. m is initialized with the formula for determining the lower half of a string.

It should be noted that depending on the font used by the editor, it is sometimes difficult to distinguish reliably between *1* and *l*. To avoid this, a different name could have been chosen for *l*.

It is important to emphasize that the commas used here are not the sequence operator (which is not presented in this text)! Rather, these commas separate the declarations of the variables used within the for loop. The second part of the header is the condition that must be true for the loop statement to execute. The last part increments *i* by 1 using the pre-increment operator. The loop statement is a compound statement. This is not mandatory, since it consists of only one statement. However, as mentioned earlier, a compound statement is used as a loop statement because it is easier to read and extend. The loop statement simply calls the `std::swap<>()` function template. To use it, one of the `<algorithm>`, `<utility>`, or `<string_view>` header files must be included. `swap<>()` exchanges the values of the two variables passed as arguments. To access the individual characters, `string::at()` is used. This member function checks if the index passed to it is valid. The first character to be swapped is accessed with `s.at(i)`, the second with `s.at(1 - i - 1)`. In Tables 11 – 12 the symbol *a* (for *alternate index*, i.e. 2nd index) was chosen for this purpose. Since this value is calculated and used only once, it is acceptable to insert it directly as an expression.

Now it is time to test. The program has been compiled to an executable file called *rv*, and the first test is made with “*Hello*”. Figure 24 shows the corresponding dialog in a console window.

```
./rv
String: Hello
Reversed string = olleH
```

Figure 24: Test of `reverseString1()` with “*Hello*”

“*Hello*” is an odd-length string. Therefore, the next test is performed with a string of even length, namely “*Hi*”. Figure 25 shows the corresponding dialog.

```
./rv
String: Hi
Reversed string = iH
```

Figure 25: Test of `reverseString1()` with “*Hi*”

Finally, the program is tested with a short string, namely “*X*” (Figure 26).

```
./rv
String: X
libc++abi.dylib: terminating with uncaught exception of type std::out_of_range:
basic_string
zsh: abort      ./rv
```

Figure 26: Test of `reverseString1()` with “*X*”

This error may seem surprising. How to find the cause of it? One way is to use a debugger, another option is to send all the variables of the for loop to cout, but the easiest way is to carefully re-read the for loop with the knowledge of the reported error in mind. Then all the variables of the for loop are replaced with the actual values resulting from a string of size 1. This is called a *paper-pencil test*. Listing 50 shows this process and its result.

```

1   for (string::size_type i { 0 },
2       l { s.length() - 1 },
3       m { l / 2 - 1 - 1 };
4       i <= m; ++i)
5   {
6       swap(s.at(i), s.at(l - i - 1));
7   }

```

Listing 50: Paper-pencil test of reverseString1() function for “X”

Obviously there is a problem with initializing `m` to `-1`! If `m` were actually initialized to `-1`, the condition of the for loop, namely `0 <= -1`, would prevent the execution of the loop statement at all, and the for loop would terminate without error. So the loop statement must be executed at least once, and the error is probably caused by passing an invalid index to `s.at()`. `-1` is definitely an invalid index, but the loop statement would not execute if `m` is `-1`. So one might suspect that `string::size_type` is an unsigned integral type. If so, subtracting 1 from 0 of `string::size_type` gives a very large value. This can be easily checked by placing the statement `cout << ("X"s.length() - 2) << endl;` at the beginning of function `main()`.

This prints `18446744073709551615` in the console window, which is a very large value, and certainly not a valid index position! Of course, this value may be different with another compiler or on another platform.

The question is how to fix this error. One possibility is to adjust the initialization of `m` in the header of the for loop. `m` could be initialized with 0 for strings of size 1 or less. Another possibility is to run the for loop only for strings of size 2 or greater. A third option is to return from the function immediately, if the size of the string is less than 2. For the program in Listing 51, the last option is chosen. The `reverseString1()` function has been renamed `reverseString2()` here.

```

1 // ReturnVoidImproved by Ulrich Eisenecker, April 8, 2021
2
3 #include <iostream>
4 #include <string> // because of string
5 #include <algorithm> // because of swap()
6 using namespace std;
7
8 void reverseString2(string & s)
9 {
10     string::size_type l { s.length() };
11     if (l < 2)
12         return;
13     for (string::size_type i { 0 },
14         m { l / 2 - 1 };
15         i <= m; ++i)
16     {

```

```

17     swap(s.at(i),s.at(l - i - 1));
18     }
19 }
20
21 int main()
22 {
23     string someString { };
24     cout << "String: "
25         << flush;
26     cin >> someString;
27     reverseString2(someString);
28
29     cout << "Reversed string = "
30         << someString
31         << endl;
32 }

```

Listing 51: `ReturnVoidImproved.cpp`

To avoid calling `s.length()` more than once, the declaration and initialization of `l` has been moved from the initialization part of the for loop header to the beginning of `reverseString2()`. If the condition `(l < 2)` evaluates to true the return statement terminates the function in a regular way.

After fixing an error or making other changes to a program, it should be tested again. This shows that the strings “Hello”, “Hi”, and “X” are reversed correctly. But when entering an empty string, the program does not stop! Fortunately, this is not a bug of the program but is caused by the behavior of the input operator `>>` for string. The `>>` operator for string will skip any whitespace character entered until a non-white space character is entered. As soon as another whitespace character is entered, the `>>` operator terminates the input and returns the string just read without whitespaces.

In fact, inputs with preceding or following whitespace or input with more than one token were not tested. For “...Hello” (the dots stand here for spaces) the program answers with “olleH”. This means that leading whitespaces are skipped. For “Hello...” the program also answers with “olleH”. This supports the assumption that the processing of the input is stopped after the first whitespace. The input of “Hello Harley” results in “elloH”, which corresponds to the previously described specification of the behavior of the `>>` operator for string.

One aspect deserves to be emphasized. The `reverseString1()` function was planned with great care, and a lot of work was put into working out the algorithm. Nevertheless, the resulting function was flawed. What does this say? ***A problem and its solution in its original domain require a mapping to a program. This mapping is neither necessarily simple nor inherently bijective. Therefore, the specifics and limitations of a program must always be thoroughly analyzed and understood.*** Testing helps to evaluate the quality of this mapping in general and to identify specific limitations. For this reason, ***testing is an indispensable activity of software development and should never be neglected.***

4.3.3.2. Returning a Reference

The `reverseString2()` function must be executed as a separate statement. A new statement is required for each subsequent action involving the reverted string. This can be easily changed. Since the parameter of `reverseString2()` is passed by reference, it is possible to return a reference to that parameter as the result. The program shown in Listing 52 contains the `reverseString3()` function, which has been modified accordingly. The `main()` function of this program demonstrates an alternative use of this function.

```
1 // ReturnReference by Ulrich Eisenecker, April 8, 2021
2
3 #include <iostream>
4 #include <string> // because of string
5 #include <algorithm> // because of swap()
6 using namespace std;
7
8 string& reverseString3(string & s)
9 {
10     string::size_type l { s.length() };
11     if (l > 1)
12     {
13         for (string::size_type i { 0 },
14             m { l / 2 - 1 };
15             i <= m; ++i)
16         {
17             swap(s.at(i),s.at(l - i - 1));
18         }
19     }
20     return s;
21 }
22
23 int main()
24 {
25     string someString { };
26     cout << "String: "
27         << flush;
28     cin >> someString;
29
30     cout << "Reversed string = "
31         << reverseString3(someString)
32         << endl;
33 }
```

Listing 52: `ReturnReference.cpp`

The `reverseString3()` function specifies `string&` as the return type, i.e. it returns a reference to `string`. This means that the evaluation of `reverseString3(someString)` here is just another name for the `string someString`, which is passed as a reference to `reverseString3()`. This guarantees that the lifetime of the object returned by reference is not limited by the lifetime of the called function! **Returning a local variable as reference is a serious error.** A compiler will normally warn against this.

The implementation of the function has been changed because a simple `return` is not sufficient. Even if the size of the `string s` is smaller than 2, `s` must be returned with `return s;`. Therefore, the condition was reworded and the `for` loop was

inserted into a compound statement to follow the recommendation to always use a compound statement in such cases, even if a single statement would suffice. Since `reverseString3()` changes a reference parameter and returns a reference to that parameter, it can be used in an expression as a function call or in a statement as a stand-alone *procedure* call. Therefore, it is not appropriate to prefix its declaration with the `[[nodiscard]]` attribute.

As the `main()` function demonstrates, `reverseString3()` can now be evaluated as part of an expression, allowing a more compact formulation for sending the result to `cout`.

4.3.3.3. Returning a Value

Since all previous versions require a parameter to be passed by reference to reverse a string, it is not possible to apply it to a string literal, e.g. "Hello"s. This problem can be solved by passing a reference-to-const parameter. Listing 53 shows a program with a corresponding implementation called `reverseString4()`.

```
1 // ReturnValue by Ulrich Eisenecker, April 8, 2021
2
3 #include <iostream>
4 #include <string> // because of string
5 #include <algorithm> // because of swap()
6 using namespace std;
7
8 [[nodiscard]] string reverseString4(const string & s)
9 {
10     string result { s };
11     string::size_type l { result.length() };
12     if (l > 1)
13     {
14         for (string::size_type i { 0 },
15             m { l / 2 - 1 };
16             i <= m; ++i)
17         {
18             swap(result.at(i), result.at(l - i - 1));
19         }
20     }
21     return result;
22 }
23
24 int main()
25 {
26     cout << "Reversed string = "
27         << reverseString4("Hello"s)
28         << endl;
29 }
```

Listing 53: `ReturnValue.cpp`

The `reverseString4()` function returns a string and takes as parameter a reference-to-const of type `string` with the name `s`. Thus, it can be called with a string variable, an expression evaluating to a string, or a string literal. If necessary, a temporary string object is created.

Again, the implementation had to be adapted to these changes. First, a local string variable named `result` is defined and initialized with `s`. In this way, `result` is a value copy of `s`. Consequently, the following code was rewritten using `result`. Thus, `l` now contains the size of `result`, and `swap()` is applied to the characters of `result`. Finally, `return result;` returns the reversed string as a value. Conceptually, this is indeed a new object. In practice, C++ tacitly applies performance-related optimizations so that the number of copies of the function result involved is kept to a minimum. As `main()` demonstrates, `reverseString4()` can also be invoked with a string literal. Since the returned result is the only effect of `reverseString4()`, it is appropriate to prefix its definition with the `[[nodiscard]]` attribute. This prevents the result of the function evaluation from being silently ignored.

The `reverseString4()` function is indeed an example where passing the string `s` by value would allow strong optimization by the compiler, as mentioned in the [Call by Reference](#) Section. A closer look reveals that the third piece of advice for passing a function parameter by value (Section [Call by Reference](#)) also applies. As Listing 54 shows, passing string `s` as a value allows to simplify the function implementation as demonstrated in `reverseString5()`.

```

1 // ReturnValue2ndVersion by Ulrich Eisenecker, February 6, 2022
2
3 #include <iostream>
4 #include <string> // because of string
5 #include <algorithm> // because of swap()
6 using namespace std;
7
8 [[nodiscard]] string reverseString5(string s)
9 {
10     string::size_type l { s.length() };
11     if (l > 1)
12     {
13         for (string::size_type i { 0 },
14             m { l / 2 - 1 };
15             i <= m; ++i)
16         {
17             swap(s.at(i), s.at(l - i - 1));
18         }
19     }
20     return s;
21 }
22
23 int main()
24 {
25     cout << "Reversed string = "
26          << reverseString5("Hello"s)
27          << endl;
28 }

```

Listing 54: `ReturnValue2ndVersion.cpp`

4.3.3.4. Designing a Function

Of course, a function can also return a pointer. Also, a value and a reference can be returned as `const`, and a pointer can be returned as a reference. However, this is only useful in rare cases. In general, the following advice can be given for the design of functions:

1. Normally, a function should only take reference-to-const parameters and return a value as the result of its evaluation.
2. A function should return `void` if it is designed as a procedure. In that case, it should specify at least one reference parameter to be changed as a consequence of executing the function, unless its sole purpose is to produce a side effect, such as sending the value of its parameter(s) to `cout`.
3. When a function calculates more than one result, it may be appropriate to pass a corresponding number of reference parameters.

One restriction must be mentioned: The above advice applies only to functions as being presented so far (so-called *free functions*). It is not complete with respect to member functions, which will be presented in more detail later.

4.3.4. Parameters with Default Values

Sometimes, functions have parameters that are called very often with the same value. For example, estimating the range of a car depending on the fuel remaining in the tank and the fuel consumption, expressed as the ratio of distance traveled and fuel consumed. Listing 55 shows the prototype of a corresponding function. The only effect of this function is to return the calculated value. Therefore, it should be declared with the `[[nodiscard]]` attribute. This has been omitted for the sake of brevity.

```
1 double estimatedRange(const double & remainingFuel,  
2                       const double & consumedFuel,  
3                       const double & distanceTraveled);
```

Listing 55: Prototype of the `estimatedRange()` function

This prototype does not specify the units of its parameters and its return value. In principle, this function works for any system of units, e.g. gallons and miles or liters and kilometers, but the units of the current parameters must match when the function is called. Typically, the fuel consumption is expressed as a ratio of fuel volume to 100 distance units. *Default values for parameters* can be used to capture both cases of the function call (Listing 56).

```
1 double estimatedRange(const double & remainingFuel,  
2                       const double & consumedFuel,  
3                       const double & distanceTraveled = 100.0);
```

Listing 56: Prototype of `estimatedRange()` function with default parameters

Now, it is possible to call `estimatedRange()` with three or two parameters, for example `cout << estimatedRange(25.5, 12.0, 200.0) << endl;` or `cout << estimatedRange(25.5, 6.0) << endl;`.

In the second case, the compiler inserts the default value for the third parameter as specified in the prototype with `= 100.0`.

Two restrictions apply:

1. If there is only one function definition, it must contain all default values. If there is a function declaration, i.e. a prototype, and the function definition, all default values must be contained exclusively in the function declaration, i.e. the prototype of the function. A separate function definition should *not* contain default values for parameters.
2. Default values for parameters must be specified from right to left. Accordingly, parameters may only be omitted from right to left. It is not possible to omit an intermediate parameter.

The use of default values for parameters should be thoroughly considered. In addition, they should be explicitly documented, either in a comment or in a so-called documentation comment, from which corresponding documentation is generated automatically. Documentation comments and tools for the automatic documentation generation will be discussed later.

4.3.5. Function Overloading

In C++, functions can have the same name as long as they differ either by the number of their parameters, the type of their parameters, or both. This is called *overloading*. Listing 57 shows how the functions for estimating the remaining range are implemented by overloading. Here, the overloaded functions accept a different number of parameters. The compiler selects a particular overloaded function with respect to the number of the parameters provided by the function call.

```
1 // Overloading by Ulrich Eisenecker, April 16, 2021
2
3 #include <iostream>
4 #include <iomanip> // because of fixed and setprecision()
5
6 using namespace std;
7
8 [[nodiscard]]
9     double estimatedRange(const double & remainingFuel,
10                          const double & consumedFuel,
11                          const double & distanceTraveled);
12 [[nodiscard]]
13     double estimatedRange(const double & remainingFuel,
14                          const double & consumedFuel);
15
16 int main()
17 {
18     cout << fixed << setprecision(2)
```

```

19     << "Estimated remaining range = "
20     << estimatedRange(30.0,6.0) // arguments are liters
21     << " km\n"                // and liter per 100 km
22     << "Estimated remaining range = "
23     << estimatedRange(15.0,12.0,200.0) // arguments are liters,
24                                     // liters, and km
25     << " km"
26     << endl;
27 }
28
29 double estimatedRange(const double & remainingFuel,
30                      const double & consumedFuel,
31                      const double & distanceTraveled)
32 {
33     return remainingFuel / (consumedFuel / distanceTraveled);
34 }
35
36 double estimatedRange(const double & remainingFuel,
37                      const double & consumedFuel)
38 {
39     return estimatedRange(remainingFuel,consumedFuel, 100.0);
40 }

```

Listing 57: Overloading.cpp

Lines 8 through 14 declare both function prototypes. Unlike in the [Parameters with Default Values](#) Section, both function declarations are prefixed with the `[[nodiscard]]` attribute. In the `main()` function, each overloaded function is called and the corresponding results are sent to standard output. For appropriate formatting, the `fixed` and `setprecision()` manipulators are used. Therefore, the header file `<iomanip>` is included. The definitions of the overloaded functions follow after `main()`.

There is one interesting aspect. The `estimatedRange()` function with three parameters has a simple but functionally complete implementation. However, it does not check possible error conditions, such as negative values, or values that cause division by zero. The other `estimatedRange()` function does not have its own fully functional implementation. Instead it calls `estimatedRange()` with three parameters with 100.0 as current value for the third parameter. This is, of course, a nice design. Suppose error checking were added to `estimatedRange()` with three parameters, the other overloaded variant would not have to be changed. Their implementations would quietly benefit from the change to the implementation of the first overloaded function.

Nevertheless, it is necessary to compare it with the solution presented in the [Parameters with Default Values](#) Section. When using default parameters, only one function is required. This is more economic than maintaining two functions. Also, one of the overloaded functions – not necessarily, but in this case – is tightly coupled to the other overloaded function. This is bad because it is no longer possible to change one of them without considering the other. Moreover, if both overloaded functions had redundant implementations, the changes would have to be applied consistently to all of them. Therefore, in the present case, the design with only one

function that uses default values for parameters is preferable to the design with two overloaded functions. This is different in a situation where the name of the overloaded function expresses a close semantic relationship, while the implementations of the functions are different. For example, the operators +, -, *, and / apply to both integral and floating-point arithmetic. But for each kind of arithmetic these operators are implemented differently. Therefore, the decision between using default values for function parameters and overloading must be carefully weighed. Using both options at the same time is not allowed, since the compiler cannot decide between ambiguous function calls.

The following variants of the program in Listings 58, 59, and 60 illustrate overloading based on different types. They also demonstrate the subtleties of passing parameters by value.

```
1 // MoreOverloading_A by Ulrich Eisenecker, April 18, 2021
2
3 #include <iostream>
4
5 using namespace std;
6
7 int timesTwo(int number)
8 {
9     cout << "timesTwo(int) ";
10    return number << 1;
11 }
12
13 int main()
14 {
15     double d { 3.0 };
16     const double dc { 4.0 };
17     cout << timesTwo(1)
18          << endl
19          << timesTwo(2.0)
20          << endl
21          << timesTwo(d)
22          << endl
23          << timesTwo(dc)
24          << endl;
25 }
```

Listing 58: *MoreOverloading_A.cpp*

The program in Listing 58 defines the `timesTwo()` function. It implements a multiplication by 2 that is specialized for `int`. Shifting the bits of an `int` value one position to the left gives the same result as multiplying it by 2^1 . Shifting two positions to the left corresponds to a multiplication by 2^2 , and so on. For signed integers, this may not result in setting the MSB, and for unsigned integers, no set bit may be shifted out to the left. This function has one parameter of type `int` which is passed by value. So the function accepts anything that is of type `int` or anything that can be automatically converted to an `int`. It copies or converts this value and uses it as a local parameter. As shown in Listing 58, the function accepts an `int` literal, of course, but it also accepts a variety of `double` values as well. It also accepts `chars`, other `int` types, and any floating-point type. Normally, functions that compute

values should not produce output. Here, a message identifying the called function is sent to cout.

```
1 // MoreOverloading_B by Ulrich Eisenecker, April 18, 2021
2
3 #include <iostream>
4
5 using namespace std;
6
7 int timesTwo(int number)
8 {
9     cout << "timesTwo(int) ";
10    return number << 1;
11 }
12
13 double timesTwo(double number)
14 {
15     cout << "timesTwo(double) ";
16     return number * 2.0;
17 }
18
19 int main()
20 {
21     double d { 3.0 };
22     const double dc { 4.0 };
23     cout << timesTwo(1)
24         << endl
25         << timesTwo(2.0)
26         << endl
27         << timesTwo(d)
28         << endl
29         << timesTwo(dc)
30         << endl;
31 }
```

Listing 59: *MoreOverloading_B.cpp*

Listing 59 overloads the `timesTwo()` function for `double`. This function uses conventional multiplication to double the value of its parameter, since bit shifting is not suitable for floating-point numbers.

As the output proves, `timesTwo(int)` is now called exclusively for the `int` literal. In all other cases `timesTwo(double)` is called. To distinguish overloaded functions in written text, it is a common to specify the types of their parameters, as it is done here.

The question is now whether it is possible to further distinguish between the available parameter variants? Yes, it is possible, as the program in Listing 60 shows.

```
1 // MoreOverloading_C by Ulrich Eisenecker, April 19, 2021
2
3 #include <iostream>
4
5 using namespace std;
6
7 int timesTwo(int number)
8 {
9     cout << "timesTwo(int) ";
10    return number << 1;
11 }
12
```



```

13 double timesTwo(const double& number)
14 {
15     cout << "timesTwo(const double&) ";
16     return number * 2.0;
17 }
18
19 double timesTwo(double& number)
20 {
21     cout << "timesTwo(double&) ";
22     return number * 2.0;
23 }
24
25 double timesTwo(double* number)
26 {
27     cout << "timesTwo(double*) ";
28     return 2.0 * *number;
29 }
30
31 double timesTwo(const double* number)
32 {
33     cout << "timesTwo(const double*) ";
34     return (*number) * 2.0;
35 }
36
37 int main()
38 {
39     double d { 3.0 };
40     const double dc { 4.0 };
41     cout << timesTwo(1)
42         << endl
43         << timesTwo(2.0)
44         << endl
45         << timesTwo(d)
46         << endl
47         << timesTwo(&d)
48         << endl
49         << timesTwo(&dc)
50         << endl;
51 }

```

Listing 60: *MoreOverloading_C.cpp*

Most importantly, `timesTwo(double)` has been removed. Otherwise it would conflict with the other overloaded functions. The overloaded function `timesTwo(const double&)` is called for the double literal `2.0`. In this case, the compiler creates a copy because the `const` parameter must not be changed. Thus, the temporarily created object can be discarded after the function call is completed. The overloaded function `timesTwo(double&)` is called for the double variable `d`. In principle, `d` could be changed within the function. If `timesTwo(double&)` were not present, `timesTwo(const double&)` would be called for `d` as well. This can be easily demonstrated by turning `timesTwo(double&)` into a comment. The `timesTwo(double&)` function is included here only for completeness. Without the requirement to actually change the parameter, this function would not exist.

Two more overloaded variants have been added, namely for a pointer to double, and a pointer to const double. As the output shows, these functions are called for the address of a double variable and the address of a const double variable. The implementation of `timesTwo(double* number)` shows an interesting detail, namely

return 2.0 * *number. The first occurrence of * refers to the multiplication operator and the second occurrence refers to the dereference operator. **Symbols and keywords that change their meaning depending on the context contribute to the difficulty of writing and reading programs in C++.** Therefore, it is important to provide additional clues to the human programmer so that they can more easily recognize what is meant by expressions and statements. For this reason, *number is enclosed in parentheses in timesTwo(const double*), although this is not necessary for the compiler. Another detail is that the expression is now changed back to (*number) * 2.0 again. For many people it is more natural to read *return number times two* than *return two times number*.

There are other possibilities of overloading based on the parameter type, which are not discussed here.

4.3.6. Specifying Pointers, References, and Constness

Before the program shown in Listing 60, the asterisk for declaring a pointer was placed midway between the type and the variable, e.g. int * ip, and surrounded by spaces. The same was done with the ampersand when defining a reference or specifying a reference parameter, e.g. string & s. Syntactically, there is no difference between placing the asterisk or ampersand immediately after the type, before the variable name, or between any number of whitespaces. The question is: Which formatting is the most appropriate? The program in Listing 61 provides some arguments.

```
1 // AsteriskAndAmpersand.cpp by Ulrich Eisenecker, April 20 2021
2
3 int main()
4 {
5     int * pointerToInt { nullptr }, anInt { 42 };
6
7     double d { 99.0 };
8     double & referenceToDouble { d }, aDouble { 100.0 };
9 }
```

Listing 61: AsteriskAndAmpersand.cpp

Line 5 contains two definitions with initializations of variables in one statement. The first one is preceded by an asterisk, the second one is not. So the first is a pointer to int, and the second is an ordinary int variable. That is, the asterisk is significant only for the immediately following variable name, but not for all other variable names that follow. This can be easily checked by swapping the initializations, namely int * pointerToInt { 42 }, anInt { nullptr };. Now the compiler reports two errors regarding invalid initialization values.

Line 8 contains similar code, but now an ampersand is used to declare a reference to double. Again, the ampersand is only relevant for the following name, but not for further following names. The check is even simpler, since it is sufficient to replace d

in the reference initialization with a literal, for example, 100.0. In this case, the compiler issues an error because it cannot initialize a reference with a temporary object.

This is an argument to put the asterisk or ampersand immediately before the name of the variable for which it is relevant. Listing 62 shows how lines 5 and 8 will then look.

```
1 /* 5 */ int *pointerToInt { nullptr }, anInt { 42 };
2 /* 8 */ double &referenceToDouble { d }, aDouble { 100.0 };
```

Listing 62: ** and & placed immediately before the variable name*

If the next name is also to be a pointer or reference, the corresponding symbol must again be placed immediately before it, for example `int *ip1, *ip2;`.

Otherwise, mixing declarations of pointers or references with normal variables could be perceived as confusing. In addition, all information relevant to an identifier can be found in exactly one place which is easier to read.

Therefore, pointers or references should be declared only once per declaration statement, and the asterisk or the ampersand should be placed immediately after the type. Listing 63 shows how lines 5 and 8 would then look.

```
1 /* 5 */ int* pointerToInt { nullptr };
2 /* 5 */ int anInt { 42 };
3 /* 8 */ double& referenceToDouble { d };
4 /* 8 */ double aDouble { 100.0 };
```

Listing 63: *& and * placed immediately after the type*

Although more verbose, this formatting will be the used from now on. It does not preclude defining more than one ordinary variable in a single definition statement, for example, `int i, j, k;`. Later, aliases are introduced for type definitions. They allow multiple pointers or references to be declared in a single statement without using asterisks or ampersands.

It is worth mentioning that this problem does not occur when specifying function parameters, since each function parameter must be specified on its own.

With `const`, the situation is somewhat different. As the program in Listing 64 illustrates, using `const` in a definition affects all subsequent variables. Thus, placing `const` after the type might give the impression that it refers only to the immediately following name. Also, `const` is an important property. It seems appropriate to emphasize it by prepending it to a definition. It should be noted that the program shown in Listing 64 is not compilable.

```
1 // EffectOfConst.cpp by Ulrich Eisenecker, April 20 2021
2
3 int main()
4 {
5     const int ic1 { 1 }, ic2 { 2 };
6     ic1 = 3; // Error, ic1 is const
```

```
7   ic2 = 4; // Error, ic2 is const
8 }
```

Listing 64: *EffectOfConst.cpp*

4.3.7. Recursion

To complete this introduction to function basics, some more details about recursive functions must be given. They have already been mentioned in the [Turing-Completeness](#) Section, and the *Checksum.cpp* program in Listing 9 already showed a recursive function.

As explained earlier, a recursive function is implemented in such a way that it calls itself either directly or indirectly. The underlying principle is to break a problem into smaller, self-similar parts until an elementary part can be solved easily. The result is then returned to the caller, which uses it to complete its computation, and so on. Recursive implementations often follow mathematical definitions and provide elegant solutions.

Looking again at the mathematical definition of the factorial function from the Section [Turing-Completeness](#), shows that it actually involves two functions:

$$\begin{aligned} \mathit{factorial}(n) &= n \cdot \mathit{factorial}(n-1) \\ \mathit{factorial}(0) &= 1 \end{aligned}$$

The first function accepts as argument a natural number greater than 0, the second function defines the result for the argument 0. Unfortunately, this cannot be realized with function overloading in C++, since a function cannot be overloaded for a specific literal value. But the trivial case of computing the factorial of 0 is necessary to terminate the recursion. Otherwise, calling the first function would lead to an infinite recursion. For this reason, a recursive function must provide termination. Usually, an if statement checks in its condition whether the termination criterion is reached and performs appropriate calculations if it is true, otherwise calling itself with at least one appropriately modified argument.

In C++, a new stack frame is created for each recursive function call. This is expensive in terms of memory consumption and efficiency. Therefore, it is recommended to provide iterative function implementations in the first place, as they perform better. Exceptionally, a recursive function is acceptable if it either contributes to a better understanding of the computation it implements and serves as a reference point for working on an equivalent iterative function implementation, or if it has an extremely elegant implementation and it can be reliably predicted that its invocation will result in moderate cost in terms of memory consumption and performance.

5. User-defined Types

The C++ language core includes a large number of fundamental types, and the C++ standard library adds an overwhelming number of additional types. Still, many programs need new types that are specific to application domains. They make it easier for human programmers to read and understand the program. They can also make it possible to write more concise programs and optimizations.

Following, it will be presented how to define and use so-called *enumeration types*. Afterwards, it will be explained how to define types for grouping related data. Eventually, *abstract data types* and *classes* will be introduced, which group related data plus related operations using these data.

5.1. Enumeration Types

There are many grading systems, for example, for pupils, for students, and for PhD candidates. Table 15 shows a common grading system for doctorates in Germany.

Latin name	German name	Quantification
summa cum laude	mit Auszeichnung	< 1.0
magna cum laude	sehr gut	1.0
cum laude	gut	2.0
rite	bestanden	3.0
non sufficit	nicht bestanden	> 3.0

Table 15: Common German Grading System for Doctorates

Interestingly, there is a great deal of agreement between universities on Latin names, but not so much on German names and especially not on quantifications. Therefore, when a program processes doctoral grades, the question arises as to which type should be used to represent grades.

One possibility would be to use `int`, where 0 stands for *summa cum laude*, 1 for *magna cum laude*, and so on. Unfortunately, the declaration `int grade;` does not prevent `grade` from being assigned the values -1 or 5. Furthermore, it is not obvious that the variable named `grade` really and exclusively represents doctoral grades.

Using the type `char` with the values 'A', 'B', 'C', etc. would only be an obfuscation of the previous problem. Worse, these values are not in the table at all.

An interesting alternative would be to use the type `std::string` to represent the grades "summa cum laude"s, "magna cum laude"s, and so on. Unfortunately, the type `std::string` is no more specific than `int` or `char`, and it is still possible to

assign incorrect values. Moreover, the lexicographic comparison "summa cum laude"s < "magna cum laude"s is not evaluated as true.

Enumeration types and enumerators provide an elegant solution to this problem. The program shown in Listing 65 demonstrates how to define the enumeration type PhDGrade (an abbreviation of *Philosophy Doctor Grade*) and its enumerators.

```
1 // Grades.cpp by Ulrich Eisenecker, April 23, 2021
2
3 #include <iostream>
4 #include <string> // because of string
5 using namespace std;
6
7 enum class PhDGrade
8 {
9     non_sufficit,
10    rite,
11    cum_laude,
12    magna_cum_laude,
13    summa_cum_laude
14 };
15
16 [[nodiscard]] string to_string(const PhDGrade& grade)
17 {
18     string name { };
19     switch (grade)
20     {
21         using enum PhDGrade;
22         case summa_cum_laude: name = "summa cum laude"s; break;
23         case magna_cum_laude: name = "magna cum laude"s; break;
24         case cum_laude       : name = "cum laude"s; break;
25         case rite            : name = "rite"s; break;
26         case non_sufficit    : name = "non sufficit"s; break;
27     }
28     return name;
29 }
30
31 int main()
32 {
33     PhDGrade alexGrade { PhDGrade::rite },
34             jordansGrade { PhDGrade::summa_cum_laude };
35     cout << boolalpha
36          << "Is Alex' grade better than Jordan's grade? "
37          << (alexGrade > jordansGrade)
38          << endl;
39
40
41     alexGrade = PhDGrade::cum_laude;
42     jordansGrade = PhDGrade::non_sufficit;
43
44     cout << "alexGrade = "
45          << to_string(alexGrade)
46          << endl
47          << "jordansGrade = "
48          << to_string(jordansGrade)
49          << endl;
50 }
```

Listing 65: Grades.cpp

The declaration of an enumeration type starts with enum class followed by the name of the enumeration type, here PhDGrade. Other possibilities are enum struct

or enum, which are not explained here. Between a pair of curly braces all enumerators are listed, separated by commas. There is no comma after the last enumerator. The declaration must be terminated by a semicolon after the closing curly brace. Here the enumerators for the 5 grades were defined in reverse order. The reason for this will be explained in a moment. It worth mentioning that the identifiers in this example were chosen to be close to the application domain. Most style guides suggest a different formatting for enumeration types and enumerators.

In the `main()` function the variables `alexGrade` and `jordansGrade` are defined as `PhDGrade` and initialized with the enumerators `PhDGrade::rite` and `PhDGrade::summa_cum_laude`. All enumerators of an enumeration type declared with `enum class` must be preceded by the name of the enumeration type, followed by the scope operator, `::`. As line 37 shows, enumerators or variables of enumeration types can be compared using comparison operators. Internally, each enumerator of a specific type is assigned an integer value, starting with 0, which is incremented by 1 for each subsequent enumerator. For this reason `PhDGrade` enumerators have been defined in reverse order, starting with `PhDGrade::non_sufficit` with implicit value 0 up to `PhDGrade::summa_cum_laude` with implicit value 4.

After that, new values are assigned to the two variables. To get a meaningful representation that can be sent to the output, for example, the `to_string()` function is called. C++ has many overloaded functions `to_string()`, but none of them accepts a parameter of type `PhDGrade`.

Therefore, an overloaded version of `to_string()` is defined which takes a `PhDGrade` as parameter. The `to_string(const PhDGrade&)` function is implemented in an easy to understand, but not necessarily elegant way. Since the only effect of calling this function is to return a `string` representation for its parameter, it is prefixed with the `[[nodiscard]]` attribute. First, the `string` variable name is defined and initialized to an empty string. Then, a new control structure is used for the first time, namely a *switch statement*. Unlike `if` and `if else`, `switch` can have multiple branches. A `switch` statement begins with the keyword `switch` followed by an expression surrounded by a pair of parentheses. Here the expression simply consists of the local variable `grade`. All branches of a `switch` statement are enclosed in curly braces. Each branch begins with the keyword `case`, followed by a literal of the same type as the `switch` expression, and a colon. The colon is followed by a statement, which must usually be terminated with `break`. If the value of the expression is equal to the literal of a case, the corresponding branch statement is executed. If a branch is not terminated it *falls through*, i.e. the next branch is executed, regardless of its case literal. There can also be a default branch. It starts with the keyword `default` followed by a colon. The cases of a `switch` statement do not have to be in any particular order. However, it is recommended to arrange them systematically. A default case is usually placed at the end of a `switch` statement.

Each case should be terminated with `break`. The fall-through option should generally be avoided.

One more detail needs to be explained, namely the statement using `enum PhDGrade;`, which precedes the first case of the switch statement. It allows the use of the enumerators without the need to prepend the enumeration type and the scope operator in the scope of the current block. Of course, it could have been the first statement of function `to_string(const PhDGrade&)`. But it is a good practice to provide a short form for accessing names only in the smallest necessary scope.

Up-to-date versions of C++ compilers should compile the program with the option `-std=c++20` without problems.

Much more could be written about enumerations. For this text it is recommended to use them only in the way they were introduced before.

5.2. Structured Datatypes

Rational numbers are used for the introduction of *structured data types*. Therefore, some basic information about rational numbers is given below. Relevant information can be found, for example, on the *World Wide Web* (“Rational Number,” 2021). The C++ standard library provides support for representing and processing rational numbers at compile time. This section is about the use of rational numbers at run time.

A rational number can be expressed as the quotient of two integers, namely the *numerator* and the *denominator*. The value range of the numerator includes all integer values including negative values and zero. As an additional requirement, the value range of the denominator can be restricted to integer values greater than zero.

A rational number is *irreducible* if the numerator and denominator have no *common divisor*. This is also called the *canonical form* of the rational number. The *Euclidean algorithm* can be used to calculate the *greatest common divisor* (abbr. *gcd*) of two integers. Reducing a rational number by the gcd of its numerator and denominator gives its canonical form. Furthermore, it can be required that rational numbers can only be created in their canonical form.

To add or to subtract two rational numbers, they must have the same denominator. If necessary, the denominators can be made equal by expanding the first rational number with the denominator of the second rational number and the second rational number with the denominator of the first rational number.

The formula for the addition is $\frac{a}{b} + \frac{c}{d} = \frac{a \cdot d + b \cdot c}{b \cdot d}$.

The formula for the subtraction is $\frac{a}{b} - \frac{c}{d} = \frac{a \cdot d - b \cdot c}{b \cdot d}$.

Multiplication of rational numbers is done by multiplying the numerators and multiplying the denominators. The formula is $\frac{a}{b} \cdot \frac{c}{d} = \frac{a \cdot c}{b \cdot d}$.

Division requires that b, c, and d are non-zero: $\frac{\frac{a}{b}}{\frac{c}{d}} = \frac{a \cdot d}{b \cdot c}$.

Any of the above operations can yield a reducible rational number.

5.2.1. Rational Numbers With Fundamental Datatypes

First, a comprehensive implementation of the rational numbers and arithmetic operations described above is presented and discussed using the programming concepts presented so far. Listing 66 shows the corresponding program. It consists of ten functions including main(). For this reason, so-called *documentation comments* are used the first time. A *documentation generator* will process these documentation comments and automatically generate a documentation. In this case *Doxygen* is used as a documentation generator. The documentation comments are therefore in a format suitable for *Doxygen*. Each comment describes the purpose of the function, its parameters and its return value.

```
1  /*! \file RationalNumberSimple.cpp
2  *
3  * Implements arithmetic for rational numbers
4  * represented by integral variables.
5  *
6  * \author Ulrich Eisenecker
7  * \date December 4, 2023
8  */
9
10 #include <iostream>
11 #include <cstdint> // Because of intmax_t.
12 #include <cmath> // Because of abs().
13 #include <numeric> // Because of gcd().
14
15 using namespace std;
16
17 /*! Returns -1 if n < 0, 0 if n == 0, +1 if n > 0.
18 */
19 [[nodiscard]] intmax_t sign(const intmax_t& n);
20
21 /*! Normalizes rational number a,b,
22 * i.e., canonical form and b > 0.
23 */
24 void normalize(intmax_t& a, intmax_t& b);
25
26 /*! Inputs rational number a,b from cin.
27 */
28 void inputRationalNumber(intmax_t& a, intmax_t& b);
29
30 /*! Outputs rational number a,b to cout.
```

```

31  */
32 void outputRationalNumber(const intmax_t& a,
33                          const intmax_t& b);
34
35 /*! Adds rational numbers a,b and c,d,
36 * and puts the result in e,f.
37 */
38 void add(const intmax_t& a,const intmax_t& b,
39         const intmax_t& c,const intmax_t& d,
40         intmax_t& e, intmax_t& f);
41
42 /*! Subtracts rational number c,d from a,b,
43 * and puts the result in e,f.
44 */
45 void subtract(const intmax_t& a,const intmax_t& b,
46             const intmax_t& c,const intmax_t& d,
47             intmax_t& e, intmax_t& f);
48
49 /*! Multiplies rational numbers a,b and c,d,
50 * and puts the result in e,f.
51 */
52 void multiply(const intmax_t& a,const intmax_t& b,
53             const intmax_t& c,const intmax_t& d,
54             intmax_t& e, intmax_t& f);
55
56 /*! Divides rational number a,b by c,d,
57 * and puts the result in e,f.
58 */
59 void divide(const intmax_t& a,const intmax_t& b,
60            const intmax_t& c,const intmax_t& d,
61            intmax_t& e, intmax_t& f);
62
63 /*! Executes each function at least once.
64 */
65 int main()
66 {
67     cout << "Helper functions ..." << endl;
68     intmax_t m { }, n { };
69     cout << "Enter m: ";
70     cin >> m;
71     cout << "Enter n: ";
72     cin >> n;
73     cout << "Sign of " << m << " = " << sign(m) << endl;
74     cout << "Sign of " << n << " = " << sign(n) << endl;
75     normalize(m,n);
76     cout << "Canonical form = ";
77     outputRationalNumber(m,n);
78
79     cout << "\n\nRational number arithmetics ..."
80          << endl;;
81     intmax_t a { }, b { }, c { }, d { }, e { }, f { };
82     cout << "Enter 1st rational number\n";
83     inputRationalNumber(a,b);
84     cout << "\nEnter 2nd rational number\n";
85     inputRationalNumber(c,d);
86     add(a,b,c,d,e,f);
87     cout << "sum = ";
88     outputRationalNumber(e,f);
89     cout << endl;
90     subtract(a,b,c,d,e,f);
91     cout << "difference = ";
92     outputRationalNumber(e,f);
93     cout << endl;
94     multiply(a,b,c,d,e,f);
95     cout << "product = ";
96     outputRationalNumber(e,f);
97     cout << endl;

```

```

98     divide(a,b,c,d,e,f);
99     cout << "quotient = ";
100    outputRationalNumber(e,f);
101    cout << endl;
102 }
103
104 intmax_t sign(const intmax_t& n)
105 {
106     if (n < 0)
107     {
108         return -1;
109     }
110     if (n > 0)
111     {
112         return +1;
113     }
114     return 0;
115 }
116
117 void normalize(intmax_t& a,intmax_t& b)
118 {
119     intmax_t divisor { gcd(a,b) };
120     a = sign(a) * sign(b) * abs(a) / divisor;
121     b = abs(b) / divisor;
122 }
123
124 void inputRationalNumber(intmax_t& a, intmax_t& b)
125 {
126     cout << "numerator: " << flush;
127     cin >> a;
128     do
129     {
130         cout << "denominator: " << flush;
131         cin >> b;
132         if (b == 0)
133         {
134             cerr << "Error, denominator may not be 0!"
135                 << endl;
136         }
137     } while (b == 0);
138 }
139
140 void outputRationalNumber(const intmax_t& a,
141                          const intmax_t& b)
142 {
143     cout << '('
144          << a
145          << '/'
146          << b
147          << ')'
148          << flush;
149 }
150
151 void add(const intmax_t& a,const intmax_t& b,
152         const intmax_t& c,const intmax_t& d,
153         intmax_t& e, intmax_t& f)
154 {
155     e = a * d + b * c;
156     f = b * d;
157     normalize(e,f);
158 }
159
160 void subtract(const intmax_t& a,const intmax_t& b,
161             const intmax_t& c,const intmax_t& d,
162             intmax_t& e, intmax_t& f)
163 {
164     e = a * d - b * c;

```

```

165     f = b * d;
166     normalize(e,f);
167 }
168
169 void multiply(const intmax_t& a,const intmax_t& b,
170             const intmax_t& c,const intmax_t& d,
171             intmax_t& e, intmax_t& f)
172 {
173     e = a * c;
174     f = b * d;
175     normalize(e,f);
176 }
177
178 void divide(const intmax_t& a,const intmax_t& b,
179            const intmax_t& c,const intmax_t& d,
180            intmax_t& e,intmax_t& f)
181 {
182     e = a * d;
183     f = b * c;
184     normalize(e,f);
185 }

```

Listing 66: *RationalNumberSimple.cpp*

Including the comments, the program comprises almost 200 lines of code. To improve readability, prototypes and function definitions have been split. All function definitions appear after the implementation of `main()`.

In the following, the rationale of each function and its implementation is discussed.

To obtain the canonical form of a rational number, its numerator and its denominator have to be divided by its *gcd* value. The `<numeric>` header file declares the `std::gcd()` function for exactly this purpose.

Interestingly, the C++ standard library does not provide a function to calculate the *sign* of a number. Therefore, the `sign()` function, which takes a parameter of type `intmax_t` as a reference to `const`, and returns a value of the type `intmax_t`, is implemented according to (“Sign Function,” 2021). One implementation detail should be mentioned. The two `if` statements check whether `n` is negative or positive and return the corresponding value. Thus, only one alternative remains, namely `n` being zero, which does not need to be checked separately. Therefore, the last statement is simply `return 0;`. The declaration of `sign()` is preceded by the `[[nodiscard]]` attribute, since returning the value is the only effect the function produces.

The `normalize()` function takes two reference parameters representing the nominator and denominator of a rational number. It serves two purposes. First, it divides the numerator and denominator by their *gcd*, reducing the rational number to its canonical form. Second, it forces the denominator to be positive. That is, either a negative sign of the denominator is shifted to the numerator or a negative sign of both is reduced.

The `inputRationalNumber()` function reads values for the numerator and denominator from `cin` and stores them into the corresponding parameters passed as references. The reading of the denominator is repeated until a value other than 0 is input.

The `outputRationalNumber()` function sends the values of the two parameters passed as reference to `const`, representing numerator and denominator, to `cout`. Both components are enclosed in a pair of parentheses and separated by a slash. This is, of course, not the same format used for input. This is an exceptional case here, since otherwise it would be easy to make a semantically invalid input, i.e., a denominator with a value of zero. ***Normally the format for input and output should be identical.***

The remaining four functions, `add()`, `subtract()`, `multiply()`, and `divide()`, have some similarities. Each function takes six parameters. The first four parameters are passed as references to `const`. They represent two rational numbers, each with numerator and denominator, to which the corresponding operation is applied. The last two parameters are passed by reference. They represent the resulting rational number with its numerator and denominator. The above formulas are implemented directly. Since each operation can result in a reducible rational number, `normalize()` is called for the resulting numerator and denominator.

The function `main()` consists of two parts. In its first part, each auxiliary function is called. In its second part, every function that deals with rational numbers is called. Despite the fact that function `main()` calls every other function directly at least once, this is not sufficient for testing. Separate and more comprehensive tests were written and executed for each function, which are not shown here.

The actual design of the functions deserves some criticism.

The `sign()` function is not specific to rational numbers. Therefore it is appropriate that it is named without explicit reference to rational numbers. That is different for the `inputRationalNumber()` and `outputRationalNumber()` functions. Their names obviously have a reference to rational numbers. Simply calling these functions `input()` and `output()` is not appropriate. The functions `normalize()`, `add()`, `subtract()`, `multiply()`, and `divide()` are specific to rational numbers, but do not express this in their names. This is inconsistent and could be considered deficient, especially since these functions do not accept very specific parameter types that makes them good candidates for overloading.

The `normalize()` function can easily be called with parameters that do not belong to a rational number. No action is taken if the function is called with invalid values, e.g. if both parameters have the value zero. Also, the parameters may be passed in the wrong order, especially if their names do not specify a particular order. This argument also applies to the other functions `add()`, `subtract()`, `multiply()`, and

divide(). It is easy to confuse the order of the six parameters, there are no mechanisms to protect against illegal values, and the interface of any function with six parameters is terribly long and not easy to use.

Structures support the definition of new data types that solve at least some of these problems. They will be discussed in the next section.

5.2.2. Documentation Generation

Last but not least, the use of a documentation generator and documentation comments to automatically generate documentation is briefly explained.

To be able to follow the following description, *Doxygen* (<https://www.doxygen.nl>) and *GraphViz* (<https://www.graphviz.org>) must be installed and their installation paths must be included in the respective system variables. It must be emphasized that downloading software from the Internet and installing it can be dangerous. Appropriate measures should therefore be taken, e.g. the download source should be credible, it should be checked that the download has not been tampered with, and it should be checked that it is free of malware.

Doxygen can be used to document many types of entities of C++ programs. Listing 66 documents only two of them, the source file itself and the functions. *Doxygen* supports a variety of formats for documentation documents. Here, the so-called *Qt-style* is used. In this style, a documentation comment starts with `/*!` and ends with `*/`. `\file`, `\author`, and `\date` are so-called *documentation commands*. As their name suggest, they set the name of the file to be documented, its author, and a date.

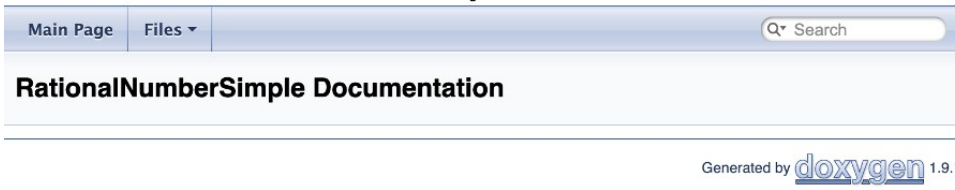
Normally, a documentation comment comes immediately before the documented entity. In the case of a file, there is no place before it. Therefore, the `\file` command is used to specify the name of the processed file.

The remaining documentation comments are all placed immediately before the documented entity, in this case a function. Each comment briefly informs about the purpose of the function and its parameters.

A special configuration file contains all the information *Doxygen* needs to run. A sample configuration file can be created by *Doxygen* itself, by typing `doxygen -g` on a command line, or by running *DoxyWizard*, a graphical user interface for creating configuration files and running *Doxygen*. In either case, a configuration file of more than 2.000 lines is created. It contains a wealth of configuration options and related information in comments.

Fortunately, the configuration file can essentially be reduced to a few options. The configuration file shown in Listing 67 generates the documentation that will be presented next.

RationalNumberSimple



```
10 DIRECTORY_GRAPH           = YES
```

Listing 67: *RationalNumberSimple_Doxyfile*

In the first line, the hash stands for a comment line which is not processed by *Doxygen*. In line 2 the name of the project is defined. Normally a project consists of several files. Here the project name corresponds to the name of the single file, but without the extension *.cpp*. Line 3 specifies the directory to which the output will be written. If the directory *RationalNumberSimple_doc* does not exist, *Doxygen* will create it in the active directory. Before starting a new *Doxygen* run, it is recommended to completely delete the contents of the directory containing the generated documentation. This ensures that parts of the documentation, i.e. files that are no longer generated, are deleted.

Line 4 specifies the directory that contains the files to be processed by *Doxygen*. If no file name is specified, all source files in this directory are processed. If a file name is specified, only that file will be processed. This is the reason why *RationalNumbersSimple.cpp* is specified here. Here it is assumed that the file to be processed exists in the active directory.

Line 5 suppresses the generation of additional documentation in *LaTeX* format, which is enabled by default. The default for documentation in *html* format is YES, so it has been omitted from the configuration file.

The *HAVE_DOT* option must be set to YES (line 6) for *GraphViz dot command* to be used. The remaining options cause the generation of various useful diagrams for which *GraphViz* is needed.

After running *Doxygen RationalNumberSimple_Doxyfile* in a terminal window, *Doxygen* creates the directory *RationalNumberSimple_doc* in the specified location. This directory contains more than 120 files in a subdirectory named *html* and another nested subdirectory named *search*. The *index.html* file in the *html* subdirectory is the entry point for the documentation. It can normally be loaded in a web browser.

Figure 27 shows a screenshot of *index.html* in a browser window.

Figure 27: *index.html* generated by *Doxygen*

Selecting *Files* → *File List* the window shown in Figure 28. opens.

RationalNumberSimple

Main Page Files

File List

Here is a list of all documented files with brief descriptions:

- [RationalNumberSimple.cpp](#)

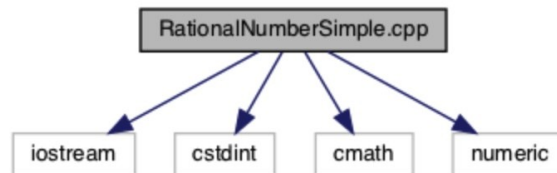
Generated by [doxygen](#) 1.9.1

Figure 28: File List generated by Doxygen

RationalNumberSimple.cpp File Reference

```
#include <iostream>
#include <cstdint>
#include <cmath>
#include <numeric>
```

Include dependency graph for RationalNumberSimple.cpp:



Functions

intmax_t	sign	(const intmax_t &n)
void	normalize	(intmax_t &a, intmax_t &b)
void	inputRationalNumber	(intmax_t &a, intmax_t &b)
void	outputRationalNumber	(const intmax_t &a, const intmax_t &b)
void	add	(const intmax_t &a, const intmax_t &b, const intmax_t &c, const intmax_t &d, intmax_t &e, intmax_t &f)
void	subtract	(const intmax_t &a, const intmax_t &b, const intmax_t &c, const intmax_t &d, intmax_t &e, intmax_t &f)
void	multiply	(const intmax_t &a, const intmax_t &b, const intmax_t &c, const intmax_t &d, intmax_t &e, intmax_t &f)
void	divide	(const intmax_t &a, const intmax_t &b, const intmax_t &c, const intmax_t &d, intmax_t &e, intmax_t &f)
int	main	()

Figure 29: File reference generated by Doxygen

Clicking on *RationalNumbersSimple.cpp* displays a relatively large document containing a reference for this file. Figure 29 shows a section of the beginning of the browser window.

The first section of the document provides information about the files included in *RationalNumberSimple.cpp*. Additionally, this information is displayed as *include-dependency graph*.

The second section displays the functions defined in RationalNumberSimple.cpp. The third section contains the information added at the beginning of RationalNumberSimple.cpp with the documentation commands `\file`, `\author`, and `\date`.

Clicking on the entry for the `main()` function in the second section, the documentation generated for it is displayed. The screenshot in Figure 30 shows the short description of `main()` and a *call graph*. The call graph of `main()` shows which other functions are called by `main()`, either directly or indirectly. Since `main()` calls every other function directly, arrow lines connect `main()` to each other function. Since there is no function that calls `main()`, there is no *caller graph* for it.

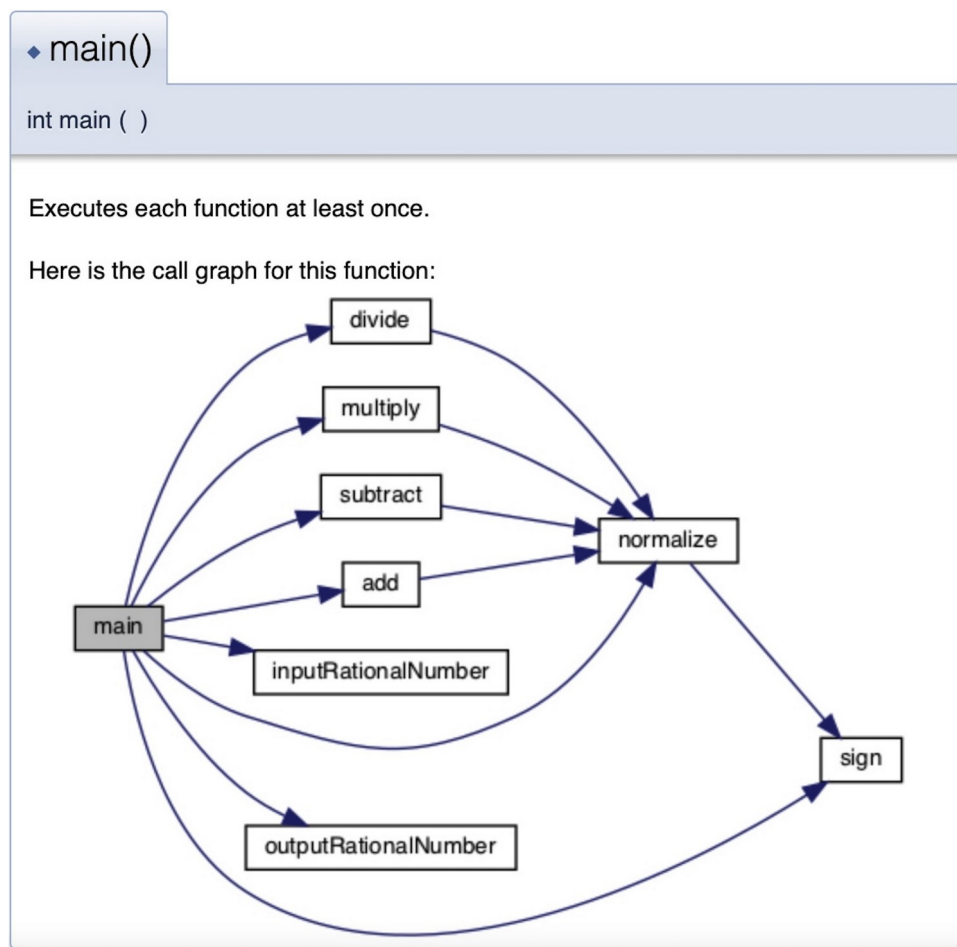


Figure 30: Call graph generated by Doxygen

Doxygen is relatively sensitive to errors and reports them only hesitantly. So if something goes wrong, it is recommended to analyze the corresponding *Doxyfile* thoroughly. In addition, documentation comments can also be a source of problems. For example, depending on the options selected, it may go unnoticed if an entity has not been commented.

5.2.3. User-defined Types for Related Data

In a rational number, the numerator and denominator are closely related. It is possible to combine both using a so-called *structure* as user-defined data type. The definition of a structure starts with the keyword `struct` followed by its name. The *members* of a structure are enclosed between a pair of curly braces. The definition of the structure must be terminated with a semicolon. Next, structures consisting only of *data members* are examined in more detail.

The structure in Listing 68 defines the `RationalNumber` data type.

```
1 struct RationalNumber
2 {
3     intmax_t numerator,
4         denominator;
5 };
```

Listing 68: *struct RationalNumber*

It has two data members, namely `numerator` and `denominator`, both of type `intmax_t`. Data members of a `struct` are defined in the same way as variables in a block. Based on the previous definition, a variable of type `RationalNumber` can be declared in the `main()` function, for example, as shown in Listing 69:

```
6 // ...
7 int main()
8 {
9     RationalNumber n;
```

Listing 69: *Declaration of a variable of RationalNumber type*

Its members can be accessed with the *member access operator* which is the dot, for example, `n.numerator`. After the declaration in Listing 69 both data members of `n` are undefined. That is, they have no meaningful value. Sending both components to `cout`, as shown in Listing 70, may result in different undefined values on each pass.

```
10 cout << n.numerator << endl;
11 cout << n.denominator << endl;
```

Listing 70: *Sending RationalNumber data members to cout*

To avoid this, it is possible to specify default values that will be used to initialize each data member, when a variable of the `struct` is defined. This is shown in Listing 71.

```
1 struct RationalNumber
2 {
3     intmax_t numerator { 0 },
4         denominator { 1 };
5 };
```

Listing 71: *Default initialization of data members in a struct*

5.2.3.1. Design Based on Reference Semantics

Now, the RationalNumberSimple.cpp (Listing 66) program is rewritten by using the struct RationalNumber instead of single integer variables. The program in Listing 72 does this in a way that closely resembles its predecessor. For example, a reference to RationalNumber is passed as an argument to the inputRationalNumber(RationalNumber&) function. This function inputs r.numerator and r.denominator directly. Each of the four arithmetic functions takes two RationalNumber parameters as references to const, combines them according to the specific arithmetic operation and places the result in the third RationalNumber parameter, which is passed as a reference.

```
1  /*! \file RationalNumberStructureReferenceSemantics.cpp
2  *
3  * Implements arithmetic for rational numbers
4  * represented by a struct
5  * using reference semantics.
6  *
7  * \author Ulrich Eisenecker
8  * \date February 8, 2022
9  */
10
11 #include <iostream>
12 #include <cstdint> // Because of intmax_t.
13 #include <cmath> // Because of abs().
14 #include <numeric> // Because of gcd().
15
16 using namespace std;
17
18 /*! Returns -1 if n < 0, 0 if n == 0, +1 if n > 0.
19 */
20 [[nodiscard]] intmax_t sign(const intmax_t& n);
21
22 /*! Represents rational number as struct.
23 */
24 struct RationalNumber
25 {
26     /*! Holds numerator of rational number.
27     * By default, numerator is initialized to 0.
28     */
29     intmax_t numerator { 0 },
30     /*! Holds denominator of rational number.
31     * By default, denominator is initialized to 1.
32     */
33     denominator { 1 };
34 };
35
36 /*! Normalizes rational number r,
37 * i.e., canonical form and r.denominator > 0.
38 */
39 void normalize(RationalNumber& r);
40
41 /*! Inputs rational number r from cin.
42 */
43 void inputRationalNumber(RationalNumber& r);
44
45 /*! Outputs rational number r to cout.
46 */
47 void outputRationalNumber(const RationalNumber& r);
48
49 /*! Adds rational numbers a and b, and puts the result in r.
```

```

50 */
51 void add(const RationalNumber& a,const RationalNumber& b,
52         RationalNumber& r);
53
54 /*! Subtracts rational number b from a, and puts the result in r.
55 */
56 void subtract(const RationalNumber& a,const RationalNumber& b,
57              RationalNumber& r);
58
59 /*! Multiplies rational numbers a and b, and puts the result in r.
60 */
61 void multiply(const RationalNumber& a,const RationalNumber& b,
62              RationalNumber& r);
63
64 /*! Divides rational number a by b, and puts the result in r.
65 */
66 void divide(const RationalNumber& a,const RationalNumber& b,
67            RationalNumber& r);
68
69 /*! Executes each function at least once.
70 */
71 int main()
72 {
73     cout << "Helper functions ..." << endl;
74     intmax_t m { }, n { };
75     cout << "Enter m: ";
76     cin >> m;
77     cout << "Enter n: ";
78     cin >> n;
79     cout << "Sign of " << m << " = " << sign(m) << endl;
80     cout << "Sign of " << n << " = " << sign(n) << endl;
81     RationalNumber o { m, n };
82     normalize(o);
83     cout << "Canonical form = ";
84     outputRationalNumber(o);
85
86     cout << "\n\nRational number arithmetics ..."
87          << endl;
88     RationalNumber a { }, b { }, c { };
89     cout << "Enter 1st rational number\n";
90     inputRationalNumber(a);
91     cout << "\nEnter 2nd rational number\n";
92     inputRationalNumber(b);
93     add(a,b,c);
94     cout << "sum = ";
95     outputRationalNumber(c);
96     cout << endl;
97     subtract(a,b,c);
98     cout << "difference = ";
99     outputRationalNumber(c);
100    cout << endl;
101    multiply(a,b,c);
102    cout << "product = ";
103    outputRationalNumber(c);
104    cout << endl;
105    divide(a,b,c);
106    cout << "quotient = ";
107    outputRationalNumber(c);
108    cout << endl;
109 }
110
111 intmax_t sign(const intmax_t& n)
112 {
113     if (n < 0)
114     {
115         return -1;
116     }

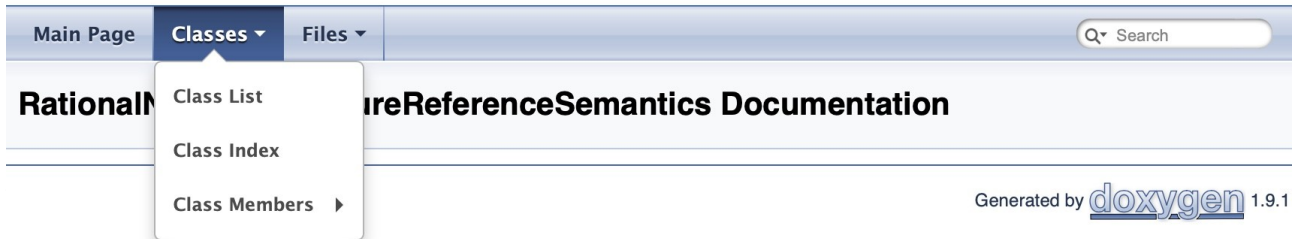
```

```

117     if (n > 0)
118     {
119         return +1;
120     }
121     return 0;
122 }
123
124 void normalize(RationalNumber& r)
125 {
126     intmax_t divisor { gcd(r.numerator,r.denominator) };
127     r.numerator = sign(r.numerator) * sign(r.denominator)
128                 * abs(r.numerator) / divisor;
129     r.denominator = abs(r.denominator) / divisor;
130 }
131
132 void inputRationalNumber(RationalNumber& r)
133 {
134     cout << "numerator: " << flush;
135     cin >> r.numerator;
136     do
137     {
138         cout << "denominator: " << flush;
139         cin >> r.denominator;
140         if (r.denominator == 0)
141         {
142             cerr << "Error, denominator may not be 0!"
143                 << endl;
144         }
145     } while (r.denominator == 0);
146 }
147
148 void outputRationalNumber(const RationalNumber& r)
149 {
150     cout << '('
151          << r.numerator
152          << '/'
153          << r.denominator
154          << ')'
155          << flush;
156 }
157
158 void add(const RationalNumber& a,const RationalNumber& b,
159         RationalNumber& r)
160 {
161     r.numerator = a.numerator * b.denominator + a.denominator * b.numerator;
162     r.denominator = a.denominator * b.denominator;
163     normalize(r);
164 }
165
166 void subtract(const RationalNumber& a,const RationalNumber& b,
167             RationalNumber& r)
168 {
169     r.numerator = a.numerator * b.denominator - a.denominator * b.numerator;
170     r.denominator = a.denominator * b.denominator;
171     normalize(r);
172 }
173
174 void multiply(const RationalNumber& a,const RationalNumber& b,
175            RationalNumber& r)
176 {
177     r.numerator = a.numerator * b.numerator;
178     r.denominator = a.denominator * b.denominator;
179     normalize(r);
180 }
181
182 void divide(const RationalNumber& a,const RationalNumber& b,
183            RationalNumber& r)

```

RationalNumberStructureReferenceSemantics



modifies it. Applying this approach to `add()`, for example, would mean that `add()` takes only two parameters, the first being a `RationalNumber` passed by reference, and the second being a `RationalNumber` passed as reference to `const`. Then `add()` performs the addition and stores the result in the first argument. Listing 73 shows a corresponding definition of `add()`.

```
1 void add(RationalNumber& a, const RationalNumber& b)
2 {
3     a.numerator = a.numerator * b.denominator + a.denominator * b.numerator;
4     a.denominator = a.denominator * b.denominator;
5     normalize(a);
6 }
```

Listing 73: *Alternative design of `add()` function*

This style of programming uses the so-called *reference semantics*, which corresponds to the imperative programming paradigm. A variable or set of variables is modified until its state corresponds to the desired solution. This style also largely corresponds to the object-oriented programming paradigm and is very popular in C++ programming.

Listing 74 shows a minimal configuration file for which *Doxygen* generates the documentation mentioned below.

```
1 # RationalNumberStructureReferenceSemantics_Doxyfile for Doxygen 1.9.1
2 PROJECT_NAME           = RationalNumberStructureReferenceSemantics
3 OUTPUT_DIRECTORY      = RationalNumberStructureReferenceSemantics_doc
4 INPUT                  = RationalNumberStructureReferenceSemantics.cpp
5 GENERATE_LATEX         = NO
6 HAVE_DOT               = YES
7 CALL_GRAPH             = YES
8 CALLER_GRAPH           = YES
9 GRAPHICAL_HIERARCHY   = YES
10 DIRECTORY_GRAPH       = YES
```

Listing 74: *RationalNumberStructureReferenceSemantics_Doxyfile*

The program shown in Listing 72 does not change the functions in terms of their tasks or their calling relationships. Generating the documentation results in exactly the same call graph for `main()`. However, the documentation generated for it contains new parts related to classes. Figures 31 and 32 show screenshots of the new menu item for classes and the section documenting `RationalNumber`.

Figure 31: *Menu entry for classes*

RationalNumberStructureReferenceSemantics

The screenshot shows the Doxygen documentation for the `RationalNumber` struct. At the top, there is a navigation bar with links for 'Main Page', 'Classes', and 'Files', along with a search box. Below the navigation bar, the title 'RationalNumber Struct Reference' is displayed. The 'Public Attributes' section lists two attributes: `intmax_t numerator { 0 }` and `intmax_t denominator { 1 }`. The 'Detailed Description' section states: 'Represents rational number as struct.' The 'Member Data Documentation' section contains two entries: 'denominator' and 'numerator'. Each entry includes its type and default value, followed by a description of its role in the struct.

Main Page | Classes ▾ | Files ▾ | Search

RationalNumber Struct Reference

Public Attributes

- `intmax_t numerator { 0 }`
- `intmax_t denominator { 1 }`

Detailed Description

Represents rational number as struct.

Member Data Documentation

◆ denominator

`intmax_t RationalNumber::denominator { 1 }`

Holds denominator of rational number. By default, denominator is initialized to 1.

◆ numerator

`intmax_t RationalNumber::numerator { 0 }`

Holds numerator of rational number. By default, numerator is initialized to 0.

Figure 32: Documentation generated by Doxygen for struct `RationalNumber`

5.2.3.2. Design Based on Value Semantics

Next, another approach is presented, which follows the so-called *value semantics*. Value semantics is a feature of the *functional programming paradigm*. Roughly speaking, it means that once an object, i.e. a variable, is created, it will not be changed. Consequently, the input of a `RationalNumber` must not change an existing variable. Instead, a new exemplar of `RationalNumber` is returned, which can be used to initialize a new variable or a `const` variable, for example. Adding two `RationalNumbers` gives a new `RationalNumber` as a result, and so on. Whenever a function returns a value, that value must be used because that is the only effect of calling that function. Therefore, all functions that return a value are prefixed with the `[[nodiscard]]` attribute.

The functions of the program shown in Listing 75 adhere to the value semantics with a few exceptions.

```

1  /*! \file RationalNumberStructureValueSemantics.cpp
2  *
3  * Implements arithmetic for rational numbers
4  * represented by a struct
5  * using value semantics.
6  *
7  * \author Ulrich Eisenecker
8  * \date February 8, 2022
9  */
10
11 #include <iostream>
12 #include <cstdint> // Because of intmax_t.
13 #include <cmath> // Because of abs().
14 #include <numeric> // Because of gcd().
15
16 using namespace std;
17
18 /*! Returns -1 if n < 0, 0 if n == 0, +1 if n > 0.
19 */
20 [[nodiscard]] intmax_t sign(const intmax_t& n);
21
22 /*! Represents rational number as struct.
23 */
24 struct RationalNumber
25 {
26     /*! Holds numerator of rational number.
27     * By default, numerator is initialized to 0.
28     */
29     intmax_t numerator { 0 },
30     /*! Holds denominator of rational number.
31     * By default, denominator is initialized to 1.
32     */
33     denominator { 1 };
34 };
35
36 /*! Returns normalized rational number r,
37 * i.e., canonical form and r.denominator > 1.
38 */
39 [[nodiscard]] RationalNumber normalize(const RationalNumber& r);
40
41 /*! Inputs rational number r from cin
42 * and returns it as value.
43 */
44 [[nodiscard]] RationalNumber inputRationalNumber();
45
46 /*! Outputs rational number r to cout.
47 */
48 void outputRationalNumber(const RationalNumber& r);
49
50 /*! Adds rational numbers a and b, and returns the result as value.
51 */
52 [[nodiscard]] RationalNumber add(const RationalNumber& a,
53                                 const RationalNumber& b);
54
55 /*! Subtracts rational number b from a, and returns the result as value.
56 */
57 [[nodiscard]] RationalNumber subtract(const RationalNumber& a,
58                                      const RationalNumber& b);
59
60 /*! Multiplies rational numbers a and b, and returns the result as value.
61 */
62 [[nodiscard]] RationalNumber multiply(const RationalNumber& a,
63                                       const RationalNumber& b);
64
65 /*! Divides rational number a by b, and returns the result as value.
66 */
67 [[nodiscard]] RationalNumber divide(const RationalNumber& a,

```



```

68         const RationalNumber& b);
69
70  /*! Executes each function at least once.
71  */
72  int main()
73  {
74      cout << "Helper functions ..." << endl;
75      intmax_t m { }, n { };
76      cout << "Enter m: ";
77      cin >> m;
78      cout << "Enter n: ";
79      cin >> n;
80      cout << "Sign of " << m << " = " << sign(m) << endl;
81      cout << "Sign of " << n << " = " << sign(n) << endl;
82      RationalNumber o { m, n };
83      o = normalize(o);
84      cout << "Canonical form = ";
85      outputRationalNumber(o);
86
87      cout << "\n\nRational number arithmetics ..."
88           << endl;
89      cout << "Enter 1st rational number\n";
90      RationalNumber a { inputRationalNumber() };
91      cout << "Enter 2nd rational number\n";
92      RationalNumber b { inputRationalNumber() };
93      cout << "sum = ";
94      outputRationalNumber(add(a,b));
95      cout << "\ndifference = ";
96      outputRationalNumber(subtract(a,b));
97      cout << "\nproduct = ";
98      outputRationalNumber(multiply(a,b));
99      cout << "\nquotient = ";
100     outputRationalNumber(divide(a,b));
101     cout << endl;
102 }
103
104 intmax_t sign(const intmax_t& n)
105 {
106     if (n < 0)
107     {
108         return -1;
109     }
110     if (n > 0)
111     {
112         return +1;
113     }
114     return 0;
115 }
116
117 RationalNumber normalize(const RationalNumber& r)
118 {
119     intmax_t divisor { gcd(r.numerator,r.denominator) };
120     RationalNumber result { };
121     result.numerator = sign(r.numerator) * sign(r.denominator)
122                     * abs(r.numerator) / divisor;
123     result.denominator = abs(r.denominator) / divisor;
124     return result;
125 }
126
127 RationalNumber inputRationalNumber()
128 {
129     RationalNumber result;
130     cout << "numerator: " << flush;
131     cin >> result.numerator;
132     do
133     {
134         cout << "denominator: " << flush;

```

```

135     cin >> result.denominator;
136     if (result.denominator == 0)
137     {
138         cerr << "Error, denominator may not be 0!"
139             << endl;
140     }
141 } while (result.denominator == 0);
142 return normalize(result);
143 }
144
145 void outputRationalNumber(const RationalNumber& r)
146 {
147     cout << '('
148         << r.numerator
149         << '/'
150         << r.denominator
151         << ')'
152         << flush;
153 }
154
155 RationalNumber add(const RationalNumber& a, const RationalNumber& b)
156 {
157     RationalNumber result;
158     result.numerator = a.numerator * b.denominator + a.denominator * b.numerator;
159     result.denominator = a.denominator * b.denominator;
160     return normalize(result);
161 }
162
163 RationalNumber subtract(const RationalNumber& a, const RationalNumber& b)
164 {
165     RationalNumber result;
166     result.numerator = a.numerator * b.denominator - a.denominator * b.numerator;
167     result.denominator = a.denominator * b.denominator;
168     return normalize(result);
169 }
170
171 RationalNumber multiply(const RationalNumber& a, const RationalNumber& b)
172 {
173     RationalNumber result;
174     result.numerator = a.numerator * b.numerator;
175     result.denominator = a.denominator * b.denominator;
176     return normalize(result);
177 }
178
179 RationalNumber divide(const RationalNumber& a, const RationalNumber& b)
180 {
181     RationalNumber result;
182     result.numerator = a.numerator * b.denominator;
183     result.denominator = a.denominator * b.numerator;
184     return normalize(result);
185 }

```

Listing 75: *RationalNumberStructureValueSemantics.cpp*

In the following, an overview of the changes is given. In addition, deviations from the strict value semantics are discussed and in some cases it is shown how these deviations can be overcome.

The `main()` function deviates in some places from the pure value semantics. First, the variables `m` and `n` are defined and initialized uniformly. Then their values are read from the standard input. That is, `m` and `n` are first declared and then assigned. Obviously, they are modified after their declaration, which is not in accordance with

the value semantics. This problem can be solved by providing a special function to input and returning an integer value. Listing 76 shows the prototype of a corresponding function.

```
1 intmax_t inputInt(const string& prompt);
```

Listing 76: inputInt() function

Of course, the function must define a local integer variable in its implementation and use it to read the input from `cin`. This cannot be circumvented in C++.

Now, `m` and `n` can be declared as `const` and initialized with the result of calling this function, as Listing 77 shows.

```
1 const intmax_t m { inputInt("Enter m: ") }, n { inputInt("Enter n: ") };
```

Listing 77: Declaration and initialization of variables by function call

Second, in the statement `o = normalize(o)`; the result is assigned to the variable `o` that was initialized in the immediately preceding statement. Again, both steps can be combined into one by using a uniform initialization, as shown in Listing 78.

```
1 const RationalNumber o { normalize({ m,n }) };
```

Listing 78: Combining normalization and initialization

Since `normalize()` expects a `RationalNumber` passed as a reference to `const`, it can be called with a corresponding initializer list, here `{ m,n }`. The corresponding temporary `RationalNumber` object is created spontaneously. The result of `normalize()` is used to initialize `o`. It should be noted that `o` is declared as `const RationalNumber` here.

The `normalize()` function takes a reference to `const` to `RationalNumber` as a parameter and returns a `RationalNumber` as value. In this way, its use conforms to value semantics, since `normalize()` does not change the parameter passed to it. The implementation of `normalize()` is different. Here, a local variable called `result` of type `RationalNumber` is defined. New values are then assigned to the members of `result`, and `result` is returned as a value.

Uniform initialization makes it possible to rewrite the implementation of `normalize()` so that it also conforms to the value semantics. The function prototype tells the compiler what type the function returns. So the compiler uses the initializer list to construct an object of that type! Of course, `divisor` can also be declared `const`. Listing 79 shows the function `normalize()` in an appropriately rewritten form.

```
1 RationalNumber normalize(const RationalNumber& r)
2 {
3     const intmax_t divisor { gcd(r.numerator,r.denominator) };
4
5     return { sign(r.numerator) * sign(r.denominator) * abs(r.numerator) / divisor,
6             abs(r.denominator) / divisor };
7 }
```

Listing 79: normalize() function rewritten

The `inputRationalNumber()` function must be used according to the value semantics. It no longer takes a parameter, but simply returns a `RationalNumber` as a value. Since it has no parameter, it can no longer be overloaded for another type because the type of a return value does not participate in the overload resolution. Its implementation also violates value semantics. It defines a local variable called `result` of type `RationalNumber`. Its members are subsequently used to input new values. This problem can be circumvented by using separate integral variables. However, in the case of the denominator, repeated changes may occur if the user specifies 0 as input. Nevertheless, `return normalize(result);` fully adheres to value semantics. It passes `result` to `normalize()`, which returns a new `RationalNumber`, which is subsequently returned as the function result. Fortunately, the compiler applies appropriate optimizations to make this procedure efficient. Incidentally, the call to `normalize()` with the rational number just read by the call to `inputRationalNumber()` is a change from the previous version in Listing 72. It may have been introduced there as well. However, in an effort to keep the changes from the `RationalNumberSimple.cpp` program (Listing 66) to a necessary minimum, this has been avoided.

In fact, `inputRationalNumber()` can be rewritten so that its implementation fully conforms to the value semantics, as shown in Listing 80. The necessary components are a uniform initialization, the `inputInt()` function presented above, and a modified do loop.

```
1 RationalNumber inputRationalNumber()
2 {
3     const intmax_t numerator { inputInt("numerator: ") };
4     do
5     {
6         const intmax_t denominator { inputInt("denominator: ") };
7         if (denominator != 0)
8         {
9             return normalize({ numerator, denominator });
10        }
11    } while (true);
12 }
```

Listing 80: InputRationalNumber() function rewritten

The do loop continues forever if no value is entered for the denominator other than 0. As soon as a valid denominator is entered, the return statement exits the function. Since the compiler knows that `normalize()` expects a `RationalNumber`, it constructs one on the fly using the supplied initializer list. It must be mentioned that `denominator` is declared as `const`. This is possible because it is valid only within the block of the do loop. It is destroyed each time the condition is checked and re-declared and re-initialized when the block of the do loop is executed again.

Each function implementing an arithmetic operation takes two reference-to-const parameters of type `RationalNumber` and returns the result as value of type

RationalNumber. Their implementations do not fully conform to value semantics because each function defines a local variable result whose members are changed before being returned. The add() function in Listing 81 shows how to change this.

```
1 RationalNumber add(const RationalNumber& a, const RationalNumber& b)
2 {
3     return normalize({ a.numerator * b.denominator +
4                       a.denominator * b.numerator,
5                       a.denominator * b.denominator });
6 }
```

Listing 81: add() function rewritten

5.3. Abstract Data Types

The programs shown in listings 72 and 75 have some similarities in terms of their functions:

- Except for main(), both programs have a function that has nothing to do with RationalNumber, namely sign(). It is an auxiliary function used in functions related to RationalNumber.
- They have three functions that take exactly one parameter of type RationalNumber or return a value of type RationalNumber, namely normalize(), inputRationalNumber() and outputRationalNumber().
- The four functions add(), subtract(), multiply() and divide() perform arithmetic operations on two RationalNumbers and return a new RationalNumber as the result of the calculation.

An *abstract data type* (abbr. *ADT*) allows the definition of a new type for related data and functions that work with them. This is called *encapsulation*. In addition, an ADT hides the details about the data and the implementations of the functions that work with them. This is referred to as *information hiding*. The only information an ADT reveals to clients is the interfaces of its functions.

5.3.1. Classes

In C++, the struct and class keywords provide support for ADTs. The difference between struct and class is that in a struct everything is public by default, while in a class everything is private by default. In the following, only class will be used because it is closer to the concept of an ADT in terms information hiding.

Both an ADT and a class serve as *blueprints* to construct so-called *exemplars*, *instances*, or *objects*. The latter three terms are all synonyms. **All objects share a common life cycle. Each object is created, used, and eventually destroyed. A class can have any number of constructors that are responsible for creating and initializing an object.** The rules for function overloading apply, i.e.

constructors must differ in the number and/or types of their parameters. **A class has exactly one destructor, which is responsible for the destruction of an object and the necessary cleanup.** All other functions of a class fall in the use of an object in its life cycle. All of these functions are called either *member functions* in C++ terminology or *methods* in the more general terminology of object-oriented programming. Listing 82 shows a rough schema for defining a class.

```
1 class SomeClass
2 {
3     public:
4         SomeClass(); // 0, 1, or more explicitly defined constructors
5         ~SomeClass(); // 0 or 1 explicitly defined destructor
6         publicMethods(); // any number
7     private:
8         privateMethods(); // any number
9         privateData; // any number
10 };
```

Listing 82: Schema for defining a class

The definition of a class begins with the keyword `class`, followed by the name of the class, which in Listing 82 is `SomeClass`. The definition is enclosed in a pair of curly braces. It is terminated by a semicolon, i.e. `class SomeClass { /* ... */ };`.

The keyword `public` followed by a colon means that all members defined after it are *accessible from outside the class*.

A *constructor* has the same name as the class. In C++ there are various kinds of constructors. One of them is the so-called *standard constructor*, which has no argument. **In some cases, if a standard constructor is not explicitly defined, the compiler generates a default implementation for it.** For this reason, the comment in line 4 in Listing 82 says *0, 1, or more explicitly defined constructors*. **A constructor is responsible for creating a fully initialized object**, that is, after an object has been constructed, it must not require any further initialization. More information on constructors will be provided as needed.

A *destructor* starts with a tilde and has the same name as the class. **A class has exactly one destructor**, which is either user-defined or automatically generated by the compiler. For this reason, the comment on line 5 in Listing 82 says *0 or 1 explicitly defined destructor*. **A destructor has no parameters. Normally, a destructor is never called directly.**

The keyword `private` followed by a colon *restricts access to all subsequent members only within the class*. Data members and auxiliary member functions that are used only within the class are normally declared `private`.

In a class, everything is – by default – `private`. Therefore, `public` is required to declare publicly accessible members. Besides `public` and `private` there is a third access specifier, `protected`. It is not explained in detail in this text.

A class can have multiple sections that are public, protected, or private. It is common for there to be only one of each section, in the exact order of public, protected, and private.

5.3.2. Simple Class Buddy

The program in Listing 83 demonstrates most of the concepts mentioned earlier. The Buddy class is very simple. It has a data member which contains the name of an object. There are methods to get and set the name, as well as a constructor and a destructor. Each method sends a message to standard output indicating which method was called.

```
1 // Buddy.cpp by Ulrich Eisenecker, May 12, 2021
2
3 #include <iostream>
4 #include <string> // Because of string.
5 using namespace std;
6
7 class Buddy
8 {
9     public:
10     Buddy(const string& n):m_name { n }
11     {
12         cout << "Buddy::Buddy(const string&)" << endl;
13     }
14     ~Buddy()
15     {
16         cout << "Buddy::~~Buddy()" << endl;
17     }
18     [[nodiscard]] const string& name() const
19     {
20         cout << "const string& Buddy::name() const" << endl;
21         return m_name;
22     }
23     void name(const string& n)
24     {
25         cout << "void Buddy::name(const string&)" << endl;
26         m_name = n;
27     }
28     private:
29     string m_name { };
30 };
31
32 int main()
33 {
34     Buddy b { "Blake" };
35     cout << b.name() << endl;
36     b.name("Taylor");
37     cout << b.name() << endl;
38 }
```

Listing 83: Buddy.cpp

As mentioned earlier, the member access operator `.` is used to access a member of an exemplar of a class, especially from outside the class, for example `b.name()` in the `main()` function. Within a class, a member of the same class is usually accessed directly. That is, the currently active object is not specified, for example `return`

`m_name`; in `Buddy::name()`. To refer to a class member independently of an object, the scope operator is used, for example, `Buddy::name()` refers to the member function `name()` of class `Buddy`. The class name cannot be omitted, since other classes may also have a member function `name()` and, in addition, a free function `name()` may exist.

The only data member of the `Buddy` class is `m_name` of type `std::string`. It is defined at the end of the class in the `private` section. There it is explicitly initialized with an empty initializer list. This causes a default initialization with an empty string. Its name starts with the prefix `m_`. This is a convention for starting the name of a data member. In principle, it could also be named `name`. However, in this case this would conflict with the overloaded member functions `Buddy::name()`. Defining a (member) function always also defines a pointer to that function with the same name. Thus, `name` without the pair of parentheses is a pointer to the (member) function `Buddy::name()`, causing a conflict with another member of the same name. The present case is even more complicated, since there are overloaded versions of `Buddy::name()`.

Now for the `public` section, which contains only methods. The first member function is the constructor. Its parameter list specifies an argument of type `const string&`, namely `n`. Next to the parameter list, after the colon, there is a *member initializer list*. This is the proper place to initialize each data member of the class. Here, the uniform initializer syntax is used to initialize `m_name` with the value of `n`. This initialization takes precedence over the direct initialization of `m_name` at the place of its declaration. This is just redundant program code, since the compiler does not perform duplicate initialization. Initialization in the member initializer list takes precedence over initialization at the point of member declaration. If more than one member is to be initialized, each must be separated from its predecessor by a comma. The body of the constructor sends a message to `cout` informing about its execution.

Because of the user-defined constructor the compiler no longer generates a standard constructor. Adding the statement `Buddy anotherBuddy;` to `main()` would result in a corresponding error message from the compiler. Nevertheless, the compiler will automatically generate a copy constructor if needed. This can be checked, for example, by providing a function `printName()` with a parameter of type `Buddy` passed by value, as shown in Listing 84.

```
1 void printName(Buddy b)
2 {
3     cout << b.name() << endl;
4 }
```

Listing 84: *Passing a Buddy exemplar by value*

Calling `printName()` in `main()`, for example, `printName(b);`, sends `b`'s name to standard output. Since `b` is passed by value, a copy is created by calling the copy

constructor of Buddy. This automatically generated copy constructor has no side effect, such as sending a message to cout. But the temporary object is destroyed when it is no longer needed. This is the reason why the call to the destructor of Buddy is displayed twice in the terminal window. In addition, printName() can also be called with a std::string literal, for example, printName("Kyle"s). This time, the compiler calls the user-defined constructor and creates a temporary exemplar of Buddy. This is possible, because b is passed by value.

There are two more public member functions, which are so-called *getter* and *setter methods* for the m_name attribute. The first overload of name(), a getter method, has no parameter and returns m_value as a reference to const. Thus, the caller of name() has access to m_value, but cannot change it. It would be also possible to return a copy of m_name, but this would consume time and memory to create. Since the only effect of calling this member function is to return the name of a Buddy exemplar, it is prefixed with the [[nodiscard]] attribute.

An important detail is that this member function is declared as const. The trailing const indicates that the call to this member function must not change the state of the receiver object. For this reason, this member function can be called even for objects declared as const or passed to another function as reference to const. In addition, it is forbidden to do anything in this member function that can change the state of the receiver object, such as calling a non-const member function.

If Buddy::name() were converted to a free function, the receiver object would have to be passed by value or as reference to const, i.e. string name(const Buddy& b), to achieve the same effect. ***It is helpful to imagine that the call to a member function always has an implicit first parameter, namely the receiver object.*** When a member function is declared const, it treats the object as if it had been declared as const for the call to that function. Any member function that is not allowed to change the state of an object should be declared as const. const also participates in the overload resolution of member functions. Two member functions can differ only in that one of them is declared const.

The second overload of name() returns nothing. It is passed a std::string as a reference to const named n. This is a setter method that assigns the parameter n to m_name. Passing the parameter n by value would also be a viable option. Whether it makes more sense to pass the parameter as a reference to const or by value requires a deeper analysis that cannot be done here. Since this method changes the state of the object for which it is called, it cannot be declared as const.

5.3.3. Class Design Based on Reference Semantics

The procedural programming paradigm assumes that a program changes the state of its variables until the desired solution is obtained. Reference semantics supports

this by allowing an object to change after it has been created and initialized. That is, with reference semantics an object can normally change its state.

This first design of class `RationalNumber` is based on reference semantics.

The `sign()` function is not directly related to `RationalNumber`. Therefore it remains a free function. All other functions that are obviously related to `RationalNumber` are made member functions of the `RationalNumber` class.

There are five groups of member functions:

1. Constructor(s): Create and initialize an exemplar of `RationalNumber` with default values 0 for numerator and 1 for denominator, a value for numerator and 1 for denominator, or values for both, numerator and denominator
2. Member functions for input and output: They read numerator and denominator of an exemplar of `RationalNumber` from standard input or write them to standard output.
3. Getters and setters for numerator and denominator: They return the corresponding member variables or assign new values to them.
4. Member functions that perform arithmetic operations: They allow addition, subtraction, multiplication and division of `RationalNumbers`. Due to reference semantics, the result of each arithmetic operation is stored in the receiver object. That is, calling an arithmetic member function changes the receiver object.
5. Auxiliary member function(s): Currently, the only member of this group is `RationalNumber::normalize()`. It is called whenever a `RationalNumber` is changed to its canonical form. Any member function that creates or modifies a `RationalNumber` must leave the object in canonical form.

The destructor generated by the compiler is sufficient. Therefore, there will be no user-defined destructor.

All member functions that do not return a specific value are declared to return the active object as a reference to `RationalNumber`. This allows further use of the returned object as will be shown later.

The complete program is shown in Listing 85. Each relevant part is documented with comments for *Doxygen*.

```
1  /*! \file RationalNumberClassReferenceSemantics.cpp
2  *
3  *  Implements arithmetic for rational numbers
4  *  represented by a class
5  *  using reference semantics.
6  *
7  *  \author Ulrich Eisenecker
8  *  \date December 14, 2023
9  */
10
11 #include <iostream>
```

```

12 #include <stdint> // Because of intmax_t.
13 #include <cmath> // Because of abs().
14 #include <numeric> // Because of gcd().
15
16 using namespace std;
17
18 /*! Returns -1 if n < 0, 0 if n == 0, +1 if n > 0.
19 */
20 [[nodiscard]] intmax_t sign(const intmax_t& n);
21
22 /*! Represents rational number as class.
23 */
24 class RationalNumber
25 {
26 public:
27     /*! Constructs a normalized RationalNumber exemplar
28     * with n for m_numerator and d for d_denominator;
29     * default value for n is 0,
30     * default value for d is 1.
31     */
32     RationalNumber(const intmax_t& n = 0, const intmax_t& d = 1):
33         m_numerator { n }, m_denominator { d }
34     {
35         normalize();
36     }
37     /*! Returns m_numerator as reference to const.
38     */
39     [[nodiscard]] const intmax_t& numerator() const
40     {
41         return m_numerator;
42     }
43     /*! Returns m_denominator as reference to const.
44     */
45     [[nodiscard]] const intmax_t& denominator() const
46     {
47         return m_denominator;
48     }
49     /*! Sets m_numerator to n,
50     * normalizes rational number,
51     * and returns *this as reference.
52     */
53     RationalNumber& numerator(const intmax_t& n)
54     {
55         m_numerator = n;
56         normalize();
57         return *this;
58     }
59     /*! Sets m_denominator to d,
60     * normalizes rational number,
61     * and returns *this as reference.
62     */
63     RationalNumber& denominator(const intmax_t& d)
64     {
65         m_denominator = d;
66         normalize();
67         return *this;
68     }
69     /*! Adds rational numbers *this and r,
70     * stores result in *this,
71     * and return *this as reference.
72     */
73     RationalNumber& add(const RationalNumber& r);
74     /*! Subtracts rational number r from *this,
75     * stores result in *this,
76     * and return *this as reference.
77     */
78     RationalNumber& subtract(const RationalNumber& r);

```

```

79     /*! Multiplies rational numbers *this and r,
80     * stores result in *this,
81     * and return *this as reference.
82     */
83     RationalNumber& multiply(const RationalNumber& r);
84     /*! Divides rational numbers *this by r,
85     * stores result in *this,
86     * and return *this as reference.
87     */
88     RationalNumber& divide(const RationalNumber& r);
89     /*! Inputs rational number from cin
90     * and returns *this as reference.
91     */
92     RationalNumber& input();
93     /*! Outputs rational number to cout
94     * and returns *this as reference.
95     */
96     RationalNumber& output();
97 private:
98     /*! Normalizes rational number,
99     * i.e., canonical form and m_denominator > 0.
100    */
101    void normalize();
102    /*! Holds numerator of rational number.
103    * By default, numerator is initialized to 0.
104    */
105    intmax_t m_numerator { 0 },
106    /*! Holds denominator of rational number.
107    * By default, denominator is initialized to 1.
108    */
109    m_denominator { 1 };
110 };
111
112 /*! Executes each free function and
113 * each member function of RationalNumber
114 * at least once.
115 */
116 int main()
117 {
118     cout << "Helper functions ..." << endl;
119     intmax_t m { }, n { };
120     cout << "Enter m: ";
121     cin >> m;
122     cout << "Enter n: ";
123     cin >> n;
124     cout << "Sign of " << m << " = " << sign(m) << endl;
125     cout << "Sign of " << n << " = " << sign(n) << endl;
126
127     cout << "\n\nRational number arithmetics ..."
128     << endl;
129     RationalNumber a { }, b { }, c { };
130     cout << "Enter 1st rational number\n";
131     a.input();
132     cout << "\nEnter 2nd rational number\n";
133     b.input();
134     c = a; // Assign a to b to save its state,
135     a.add(b);
136     cout << "sum = ";
137     a.output();
138     cout << endl;
139     a = c; // Restore saved state of a.
140     a.subtract(b);
141     cout << "difference = ";
142     a.output();
143     cout << endl;
144     a = c; // Restore saved state of a.
145     a.multiply(b);

```

```

146     cout << "product = ";
147     a.output();
148     cout << endl;
149     a.numerator(c.numerator()); // Restore saved numerator of a.
150     a.denominator(c.denominator()); // Restore saved denominator of a.
151     a.divide(b);
152     cout << "quotient = ";
153     a.output();
154     cout << endl;
155 }
156
157 intmax_t sign(const intmax_t& n)
158 {
159     if (n < 0)
160     {
161         return -1;
162     }
163     if (n > 0)
164     {
165         return +1;
166     }
167     return 0;
168 }
169
170 RationalNumber& RationalNumber::add(const RationalNumber& r)
171 {
172     m_numerator = m_numerator * r.denominator()
173                 + m_denominator * r.numerator();
174     m_denominator = m_denominator * r.denominator();
175     normalize();
176     return *this;
177 }
178
179 RationalNumber& RationalNumber::subtract(const RationalNumber& r)
180 {
181     m_numerator = m_numerator * r.denominator()
182                 - m_denominator * r.numerator();
183     m_denominator = m_denominator * r.denominator();
184     normalize();
185     return *this;
186 }
187
188 RationalNumber& RationalNumber::multiply(const RationalNumber& r)
189 {
190     m_numerator = m_numerator * r.numerator();
191     m_denominator = m_denominator * r.denominator();
192     normalize();
193     return *this;
194 }
195
196 RationalNumber& RationalNumber::divide(const RationalNumber& r)
197 {
198     m_numerator = m_numerator * r.denominator();
199     m_denominator = m_denominator * r.numerator();
200     normalize();
201     return *this;
202 }
203
204 RationalNumber& RationalNumber::input()
205 {
206     cout << "numerator: " << flush;
207     cin >> m_numerator;
208     do
209     {
210         cout << "denominator: " << flush;
211         cin >> m_denominator;
212         if (m_denominator == 0)

```

```

213     {
214         cerr << "Error, denominator may not be 0!"
215             << endl;
216     }
217     } while (m_denominator == 0);
218     normalize();
219     return *this;
220 }
221
222 RationalNumber& RationalNumber::output()
223 {
224     cout << '('
225          << m_numerator
226          << '/'
227          << m_denominator
228          << ')'
229          << flush;
230     return *this;
231 }
232
233 void RationalNumber::normalize()
234 {
235     intmax_t divisor { gcd(m_numerator,m_denominator) };
236     m_numerator = sign(m_numerator) * sign(m_denominator)
237                 * abs(m_numerator) / divisor;
238     m_denominator = abs(m_denominator) / divisor;
239 }

```

Listing 85: *RationalNumberClassReferenceSemantics.cpp*

Since the `sign()` function is the same as in Listing 72, it is not explained again.

The `RationalNumber` class has a public and a private section. First, the private section is briefly described. `RationalNumber::m_numerator` and `RationalNumber::m_denominator` are data members of `RationalNumber`, both of type `intmax_t`. They are initialized in-class with 0 and 1. Initialization via the initializer list of a constructor would take precedence over this initialization. This is indeed the case, as explained below.

The helper function `RationalNumber::normalize()` is now a private member function. As mentioned above, an object of type `RationalNumber` must be in canonical form after its creation or the execution of a possibly modifying member function. For this reason, `RationalNumber::normalize()` must be called by each of these member functions. As a consequence, a client no longer needs to call this function directly. Therefore, it has been moved into the private section.

Now to the public section.

`RationalNumber` has only one constructor. It fulfills all the above requirements. How is this possible?

The constructor has two parameters, namely `const intmax_t& n = 0` and `const intmax_t& d = 1`.

Each parameter has a default value which is 0 for `n` (the numerator), and 1 for `d` (the denominator). The rational number 0/1 is equal to 0. When calling the constructor,

parameters can be omitted from right to left. If the constructor is called with two parameters, the caller passes both components of a rational number. If the constructor is called with only one argument, it becomes a *type conversion constructor*. The type conversion constructor can be called automatically if a RationalNumber is expected and it is passed by value or as a reference to const. For example, the integer 5 is converted to a RationalNumber with 5 as RationalNumber::m_numerator and 1 as RationalNumber::m_denominator. If the constructor is called without parameter, it is used as the standard constructor. This results in a RationalNumber with 0 as RationalNumber::m_numerator and 1 as RationalNumber::m_denominator.

A few words about constructors must be added. Until C++11, a constructor whose only argument is of a different type than the class for which it is defined and which can be called implicitly was called a converting constructor. Since then, due to the availability of initializer lists, all constructors that can be called automatically are called converting constructors (*Converting Constructor - Cppreference.Com*, n.d.).

Now to the getter methods const intmax_t& RationalNumber::numerator() const and const intmax_t& RationalNumber::denominator() const. Both return the corresponding data members as reference to const. Their call must not change the state of the receiver object. For this reason, the getter methods are declared as const. Apart from returning a result, they have no other effect. Therefore, they have been prepended with the [[nodiscard]] attribute.

RationalNumber& RationalNumber::numerator(const intmax_t& n) and RationalNumber& RationalNumber::denominator(const intmax_t& d) are setter methods. They modify the receiver object. Therefore, they are not declared as const. Both return a reference to the receiver object. This way it is possible to use their result in an expression.

After the new value is assigned to the corresponding data member, RationalNumber::normalize() is called to ensure that the receiver object is subsequently in canonical form. There is a serious possibility for incorrect usage. It is possible to set RationalNumber::m_denominator to 0 by calling, for example, someRationalNumber.denominator(0);. This error is not checked in this member function. In practice, this is unacceptable, but currently, the means for error handling have not been introduced. The last statement of both setter methods is return *this;. Each member function is called for a particular object. The question is *how to refer to this specific object*. The answer is this. ***Each (non-static) member function automatically defines this as a pointer to the object that is currently being used.*** Placing the dereference operator, *, in front of the pointer gives the currently active object itself. Thus, return *this; returns the currently active object, by reference – as specified as the return type of the member function. Later, other uses of this will be presented.

In Listing 72, the functions that perform arithmetic on rational numbers have three parameters, for example `void add(const RationalNumber& a, const RationalNumber& b, RationalNumber& r);`. The first operand of the addition is `a`, its second operand is `b`, and the result is stored in `r`, which is passed by reference. In the `RationalNumber` class, the first operand is now the receiver object, i.e. `add(a, b, r)` becomes `a.add(b)`. The second operand `b` of `add(a, b, r)` becomes the first and only parameter in `a.add(b)`. The result stored in `r` in `add(a, b, r)` is subsequently stored in the receiver object. That is, `a` in `a.add(b)` corresponds to `r` in `add(a, b, r)`. Finally, the modified object is returned by `return *this;`. This allows further use of this object in an expression. None of the arithmetic methods is marked with the `[[nodiscard]]` attribute. For this reason, the result of executing any of these methods can be ignored. Because of this design, each arithmetic method modifies its receiver object. Therefore, none of them is declared as `const`.

It is worth noting that these reshaped member functions actually serve two purposes: First, they perform the corresponding arithmetic operation, and second, they assign the result to the receiver object. Therefore, one could also argue that these member functions could be renamed to `RationalNumber::assignSum()`, `RationalNumber::assignDifference()`, etc. However, for the sake of simplicity, the already introduced identifiers are kept unchanged.

The `RationalNumber::input()` member function reads new values for the receiver object's member attributes from the standard input and prevents input of 0 for `RationalNumber::m_denominator`. It also calls the `normalize()` member function. `RationalNumber::output()` sends the formatted values of the receiver object's member attributes to `cout`. Both member functions return the receiver object by reference.

It is worth noting that the definitions of the constructor, getter and setter methods appear at the point of their declaration. This has the effect that they are declared `inline`.

A function or member function is *implicitly declared inline* if its declaration is also a definition. That is, it must be reachable in the translation unit where it is accessed. An `inline` declaration suggests to the compiler that the code implementing the inline function be inserted when that function is called. Or, more explicitly, the compiler does not generate code for a function call, but inserts the code to be executed directly at the point of the function call. A compiler is not obliged to follow this suggestion. If the compiler applies inlining, it may affect code size and performance. A function can also be *explicitly declared inline* by prefixing its declaration with the keyword `inline`. In this case, the definition of the function must be also available in the same translation unit, for example, after its declaration. A concrete example for functions explicitly declared inline is not presented in this text. For more information on inlining see (*Inline Specifier - Cppreference.Com*, n.d.).

The remaining member functions of RationalNumber as well as the one free function are declared at the beginning of source file, but their definitions appear after the main() function. When a member function is defined outside its class, its name must be preceded by the name of the class followed by the scope operator, ::, for example void RationalNumber::normalize(). This indicates that the function is actually a member function and not a free function.

Some changes in main() have to be mentioned. Since a.add(b) will change the state of a, the state of a must be saved so that it can be restored afterwards. This is done by the previous assignment c = a;. After calling a.add(b) and a is output, the previous state of a is restored by the assignment a = c;. This pattern is repeated with one exception. Before calling RationalNumber::divide(), the state of a is restored by explicitly calling the corresponding getter and setter methods, so they are called at least once directly in main(). The only function that is not directly called in main() is RationalNumber::normalize(), since it is a private member of RationalNumber.

Now it is time to take a closer look at performing arithmetic operations with rational numbers. The expression to be calculated is $\frac{1}{2} \cdot \frac{1}{2} + \frac{1}{4}$.

The code snippets in Listings 86, 87, and 88 show several ways to do this. Listing 86 shows a first possibility.

```
1 RationalNumber a { 1,2 }, b { 1,2 }, c { 1,4 };
2 a.multiply(b);
3 a.add(c);
4 a.output();
5 cout << endl;
```

Listing 86: One calculation per statement

It is also possible to combine all arithmetic operations into a single expression, as shown in Listing 87. The call to a.multiply(b) returns a reference to a. Thus, the result of the call to a.add(b) can be reused immediately. By applying the dot operator, ., add() is called for the resulting object, namely a.multiply(b).add(c);. Finally, the result returned by add() is ignored.

```
1 a.multiply(b).add(c);
2 a.output();
3 cout << endl;
```

Listing 87: An expression that combines all calculations

A third possibility is shown in Listing 88. Here, a is multiplied by itself.

```
1 RationalNumber a { 1,2 }, b { 1,2 }, c { 1,4 };
2 a.multiply(a).add(c);
3 a.output();
4 cout << endl;
```

Listing 88: Multiplying a RationalNumber variable by itself

Now the expression to be calculated is varied: $\frac{1}{2} \div \frac{1}{2} + \frac{1}{4}$. Executing the code snippet shown in Listing 89 gives the expected result, namely $\frac{5}{4}$.

```
1 RationalNumber a { 1,2 }, b { 1,2 }, c { 1,4 };
2 a.divide(b).add(c);
3 a.output();
4 cout << endl;
```

Listing 89: *Dividing different RationalNumbers gives the correct result*

However, executing the code snippet shown in Listing 90 outputs $\frac{3}{4}$, which is *not* the correct result.

```
1 RationalNumber a { 1,2 }, b { 1,2 }, c { 1,4 };
2 a.divide(a).add(c);
3 a.output();
4 cout << endl;
```

Listing 90: *Dividing RationalNumber by itself gives wrong results*

The reason is that the implementation of `RationalNumber::divide()` first modifies `RationalNumber::m_numerator` and then uses this modified value to calculate `RationalNumber::m_denominator`. Why is this possible? Well, `a` is passed as a reference to `const` by the alias `r`, which prevents `r` from being modified. But `a` is also the currently active object, which is *not* `const` (just a reminder, `RationalNumber::add()` is not declared as `const`). This allows `a` to be modified, which is exactly what the design intends. This is a serious deficiency!

In fact, there are various possible fixes. One is to change only the implementation of `RationalNumber::divide()`, as shown in Listing 91.

```
1 RationalNumber& RationalNumber::divide(const RationalNumber& r)
2 {
3     RationalNumber temp { r };
4     m_numerator = m_numerator * temp.denominator();
5     m_denominator = m_denominator * temp.numerator();
6     normalize();
7     return *this;
8 }
```

Listing 91: *Improved implementation of RationalNumber::divide()*

This has the charm that it does not change the declaration of `RationalNumber`. Keeping the interface the same while changing its implementation is an application of information hiding. The ugly drawback is that it is specific to `RationalNumber::divide()` and is not motivated by functional requirements of rational number arithmetic, but simply by the specifics of the underlying design. Thus, it is not a general solution, but a specific fix that must be thoroughly documented. Otherwise, another programmer might stumble upon the seemingly deviant implementation and undo it to make it match the other implementations. In

addition, other member functions may have similar bugs. Fixing them all ad-hoc is not a good solution.

A simple fix is to change the declaration of the member function `RationalNumber::divide()` so that its parameter is passed by value (Listing 92).

```
1 RationalNumber& RationalNumber::divide(RationalNumber r)
2 {
3     m_numerator = m_numerator * r.denominator();
4     m_denominator = m_denominator * r.numerator();
5     normalize();
6     return *this;
7 }
```

Listing 92: Changing the declaration of `RationalNumber::divide()`

At a first glance, Listing 92 shows a nice solution, since it is only a small change and the implementation remains untouched. But it is a change to the interface that violates modularization.

Nevertheless, it is a good idea to apply this fix to all member functions for which it is appropriate. In general, passing `RationalNumber` exemplars by value rather than as reference to `const` becomes an overall design decision. A `RationalNumber` passed by value can never be identical to the receiver object. Consequently, a change in the receiver object can never change the state of the `RationalNumber` passed as parameter by value.

The question is how to identify such design problems and possible sources of error? One way is to thoroughly analyze the source code and try to identify such critical parts. Another is comprehensive testing, not only with different test data, but also with different arrangements of expressions and statements.

5.3.4. Class Design Based on Value Semantics

The design of the program shown in Listing 93 is based on value semantics. Besides creation of exemplars, input, and output, the other functions and member functions do not change the state of any part of the program.

```
1  /*! \file RationalNumberClassValueSemantics.cpp
2  *
3  *  Implements arithmetic for rational numbers
4  *  represented by a class
5  *  using value semantics.
6  *
7  *  \author Ulrich Eisenecker
8  *  \date December 14, 2023
9  */
10
11 #include <iostream>
12 #include <cstdint> // Because of intmax_t.
13 #include <cmath> // Because of abs().
14 #include <numeric> // Because of gcd().
15
16 using namespace std;
```

```

17
18 /*! Returns -1 if n < 0, 0 if n == 0, +1 if n > 0.
19 */
20 [[nodiscard]] intmax_t sign(const intmax_t& n);
21
22 /*! Represents rational number as class.
23 */
24 class RationalNumber
25 {
26     public:
27         /*! Constructs a normalized RationalNumber exemplar
28          * with n for m_numerator and d for d_denominator;
29          * default value for n is 0,
30          * default value for d is 1.
31          */
32         RationalNumber(const intmax_t& n = 0, const intmax_t& d = 1):
33             m_numerator { n }, m_denominator { d }
34         {
35             normalize();
36         }
37         /*! Returns m_numerator as reference to const.
38          */
39         [[nodiscard]] const intmax_t& numerator() const
40         {
41             return m_numerator;
42         }
43         /*! Returns m_denominator as reference to const.
44          */
45         [[nodiscard]] const intmax_t& denominator() const
46         {
47             return m_denominator;
48         }
49         /*! Adds rational numbers *this and r,
50          * and returns result as value.
51          */
52         [[nodiscard]] RationalNumber add(const RationalNumber& r) const;
53         /*! Subtracts rational number r from *this,
54          * and returns result as value.
55          */
56         [[nodiscard]] RationalNumber subtract(const RationalNumber& r) const;
57         /*! Multiplies rational numbers *this and r,
58          * and returns result as value.
59          */
60         [[nodiscard]] RationalNumber multiply(const RationalNumber& r) const;
61         /*! Divides rational numbers *this by r,
62          * and returns result as value.
63          */
64         [[nodiscard]] RationalNumber divide(const RationalNumber& r) const;
65         /*! Outputs rational number to cout.
66          */
67         void output() const;
68     private:
69         /*! Normalizes rational number,
70          * i.e., canonical form and m_denominator > 0.
71          */
72         void normalize();
73         /*! Holds numerator of rational number.
74          * By default, numerator is initialized to 0.
75          */
76         intmax_t m_numerator { 0 },
77         /*! Holds denominator of rational number.
78          * By default, denominator is initialized to 1.
79          */
80             m_denominator { 1 };
81 };
82
83 /*! Inputs rational number from cin

```

```

84 * and returns it as value.
85 */
86 [[nodiscard]] RationalNumber inputRationalNumber();
87
88 /*! Executes each free function and
89 * each member function of RationalNumber
90 * at least once.
91 */
92 int main()
93 {
94     cout << "Helper functions ..." << endl;
95     intmax_t m { }, n { };
96     cout << "Enter m: ";
97     cin >> m;
98     cout << "Enter n: ";
99     cin >> n;
100    cout << "Sign of " << m << " = " << sign(m) << endl;
101    cout << "Sign of " << n << " = " << sign(n) << endl;
102
103    cout << "\n\nRational number arithmetics ..."
104         << endl;
105    cout << "Enter 1st rational number\n";
106    RationalNumber a { inputRationalNumber() };
107    cout << "Numerator (a) = " << a.numerator()
108         << "\nDenominator (a) = " << a.denominator()
109         << endl;
110    cout << "Enter 2nd rational number\n";
111    RationalNumber b { inputRationalNumber() };
112    cout << "sum = ";
113    RationalNumber apb { a.add(b) };
114    apb.output();
115    cout << "\ndifference = ";
116    RationalNumber amb { a.subtract(b) };
117    amb.output();
118    cout << "\nproduct = ";
119    RationalNumber atb { a.multiply(b) };
120    atb.output();
121    cout << "\nquotient = ";
122    RationalNumber adb { a.divide(b) };
123    adb.output();
124    cout << endl;
125 }
126
127 intmax_t sign(const intmax_t& n)
128 {
129     if (n < 0)
130     {
131         return -1;
132     }
133     if (n > 0)
134     {
135         return +1;
136     }
137     return 0;
138 }
139
140 RationalNumber RationalNumber::add(const RationalNumber& r) const
141 {
142     return { m_numerator * r.m_denominator
143             + m_denominator * r.m_numerator,
144             m_denominator * r.m_denominator };
145 }
146
147 RationalNumber RationalNumber::subtract(const RationalNumber& r) const
148 {
149     return { m_numerator * r.m_denominator
150             - m_denominator * r.m_numerator,

```

```

151         m_denominator * r.m_denominator };
152     }
153
154 RationalNumber RationalNumber::multiply(const RationalNumber& r) const
155 {
156     return { m_numerator * r.m_numerator,
157             m_denominator * r.m_denominator };
158 }
159
160 RationalNumber RationalNumber::divide(const RationalNumber& r) const
161 {
162     return { m_numerator * r.m_denominator,
163             m_denominator * r.m_numerator };
164 }
165
166 void RationalNumber::output() const
167 {
168     cout << '('
169          << m_numerator
170          << '/'
171          << m_denominator
172          << ')'
173          << flush;
174 }
175
176 void RationalNumber::normalize()
177 {
178     intmax_t divisor { gcd(m_numerator,m_denominator) };
179     m_numerator = sign(m_numerator) * sign(m_denominator)
180                 * abs(m_numerator) / divisor;
181     m_denominator = abs(m_denominator) / divisor;
182 }
183
184 RationalNumber inputRationalNumber()
185 {
186     intmax_t numerator { 0 },
187             denominator { 1 };
188     cout << "numerator: " << flush;
189     cin >> numerator;
190     do
191     {
192         cout << "denominator: " << flush;
193         cin >> denominator;
194         if (denominator == 0)
195         {
196             cerr << "Error, denominator may not be 0!"
197                 << endl;
198         }
199     } while (denominator == 0);
200     return { numerator, denominator };
201 }

```

Listing 93: *RationalNumberClassValueSemantics.cpp*

This has consequences for some functions related to RationalNumber.

- Both setter methods are removed, because they would change the state of a RationalNumber.
- The member functions that perform arithmetic on rational numbers will return their result as a new exemplar of RationalNumber by value. The result of calling an arithmetic method may no longer be ignored, since it is the only

effect the method has. For this reason, each of the corresponding declarations is prefixed with the `[[nodiscard]]` attribute.

- It is not possible to have a member function `RationalNumber::input()`, since it would modify an already existing `RationalNumber`. For this reason the free function `inputRationalNumber()` takes over this task and returns a rational number.

The `sign()` function has not changed, so there is no need to explain it again. The free function `inputRationalNumber()` was justified immediately before.

The `RationalNumber` class deserves discussion.

Its private section defines the members `RationalNumber::m_numerator` and `RationalNumber::m_denominator`. The constructor calls `RationalNumber::normalize()`, a private member function to put the numerator and denominator of a rational number into canonical form. Therefore, `RationalNumber::m_numerator` and `RationalNumber::m_denominator` must be modifiable, and `RationalNumber::normalize()` cannot be declared as a `const` member function.

Now to the public part. The constructor is identical to that of `RationalNumber` in Listing 85. The same applies to the getter methods for numerator and denominator. There are two small changes to `RationalNumber::output()`. In the version with value semantics, the return value is `void`. Therefore it can be declared as a `const` member function.

All arithmetic member functions are declared as `const`, since none of them changes the state of the receiver object. Each of them uses uniform initialization to return a new exemplar of `RationalNumber` containing the result.

An interesting property of `RationalNumber` and its associated functions is that once a valid `RationalNumber` has been constructed, its state cannot be changed because all setter methods have been removed.

There are some changes in the `main()` function. The setter methods are not called because they are no longer present. A new exemplar of `RationalNumber` is defined and initialized with the result of calling an arithmetic member function of `RationalNumber`. This is necessary for *Doxygen* to create a complete call graph for `main()`. Simply writing statements like `a.add(b).output();` does not result in a corresponding link in the call graph generated by *Doxygen*.

5.3.5. Reference- vs. Value-Based Design

The programs shown in Listings 85 and 93 implement an abstract data type `RationalNumber`. Their main difference is that one program provides the ability to modify an exemplar of `RationalNumber` after it is created, while the other does not.

The version with reference-based design has two weaknesses. First, in arithmetic member functions, the same object can be both a mutable receiver object and a parameter passed as a reference to `const`. This can lead to incorrect calculations. Several applicable fixes were subsequently presented. Second, the setter method `RationalNumber::denominator()` may set `RationalNumber::m_denominator` to 0, resulting in an invalid rational number. This cannot be avoided, but this error can in principle be detected and handled. How to do this is not covered in this text.

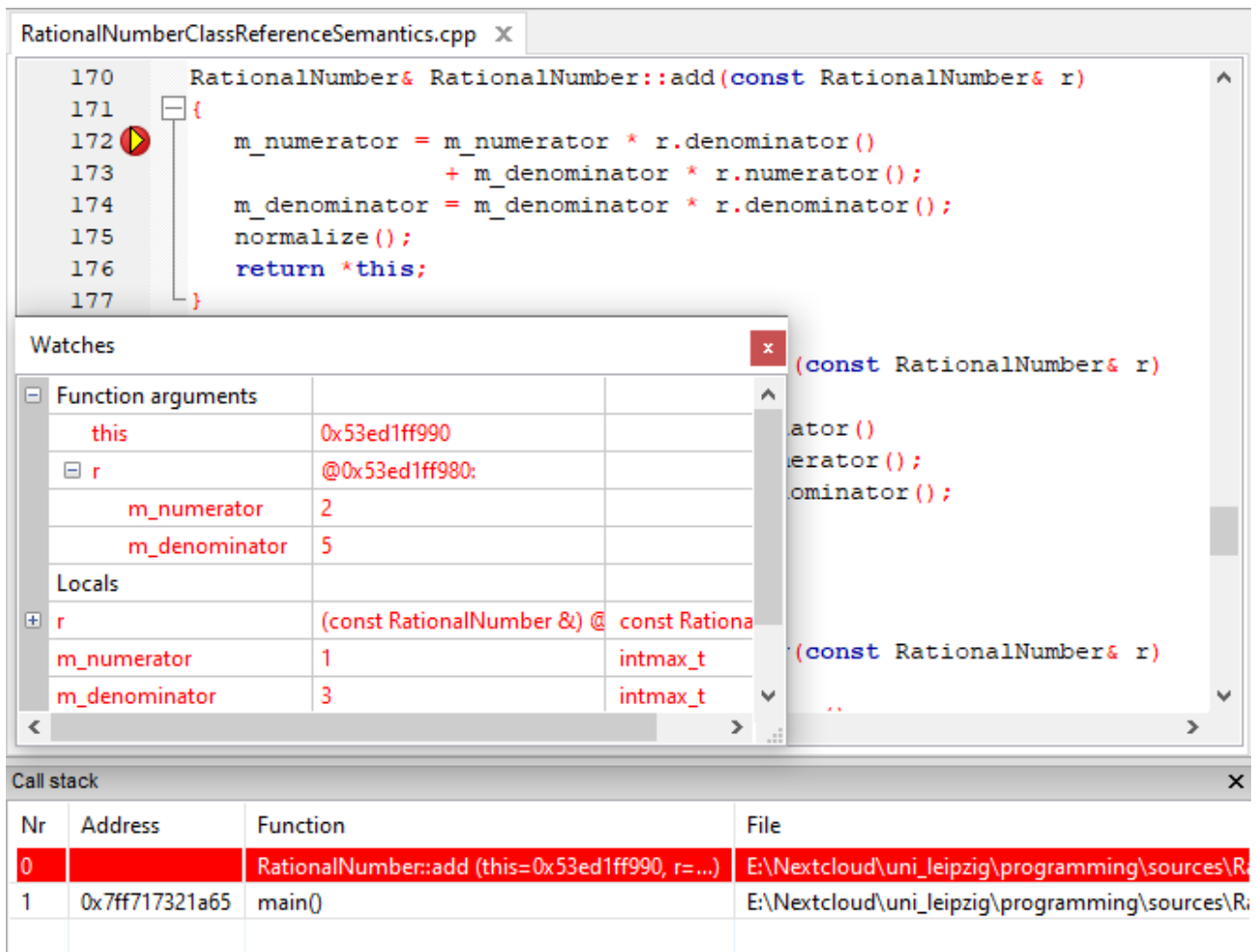
The version with value-based design also has two shortcomings. First, `RationalNumber::m_numerator` and `RationalNumber::m_denominator` cannot be declared as `const` data members because they are modified by `RationalNumber::normalize()`, which is called in the constructor to ensure the canonical form of a `RationalNumber`. An obvious workaround would be to move the necessary calculations to the initializer list of the constructor. This would result in less readable and less intentional code. The free function `inputRationalNumber()` is not turned into a member function, as this would require modifying a `RationalNumber` after it is created. In fact, it would be possible to turn it in a so-called *static member function* of `RationalNumber`. This will be explained later in Section [Overloading Operators for RationalNumber](#).

Both versions also have their own peculiarities in terms of debugging. This will be demonstrated using the member function `RationalNumber::add()`. It is called with the receiver object set to $1/3$ and the parameter object set to $2/5$.

An initial example for debugging is presented in the [Example for Debugging](#) Section. In the following it is assumed that the source program has been compiled with an option to generate debug information and that a debugger with a graphical user interface is available.

First, the reference-based design program is debugged. A breakpoint was set to the first executable line of `RationalNumber::add()`, as indicated by the red filled circle. The debugger then executed the program until the breakpoint was reached. A total of three rational numbers must be entered in a console window until program execution is paused. The value of the first rational number is not relevant. The second rational number should be $1/3$, the third $2/5$. The screenshot in Figure 33 shows the source code, the call stack and the watches after the break point was reached at the first statement.

Figure 33: Debugging the reference-based program, part 1



The bottom window in Figure 33 displays the call stack. The top function is `RationalNumber::add()`, which has been called by `main()`. The active function is highlighted by a red background. Interestingly, this line shows the value of `this`, i.e. the memory address of the receiver object, as the first parameter of this member function. This is appropriate, because `this` can be considered the implicit first parameter of any member function. The second parameter refers to `r`. Unfortunately, its memory address is not visible here. If it were visible, it would be possible to see that the two addresses are different.

The fact that the memory address of `r` is not visible in the call stack, is not problematic. The middle window shows the watches to which `m_numerator` and `m_denominator` have been added as local variables. The top lines of the Watches window show the function arguments, namely `this` and `r`, as well as the value `this` and the memory address of `r`. Different memory addresses mean different objects. So it is not one and the same object being accessed in two different ways (non-const `this` and parameter passed by reference). In the following, the data members of `r` have been expanded so their values are displayed.

Stepping over the lines of `RationalNumber::add()`, the local variables `m_numerator` and `m_denominator` change their values. This way it becomes clear how the values of the variables change incrementally.

The screenshot in Figure 34 shows the same windows, but now just before the return of `RationalNumber::add()`, as marked by the yellow filled arrowhead.

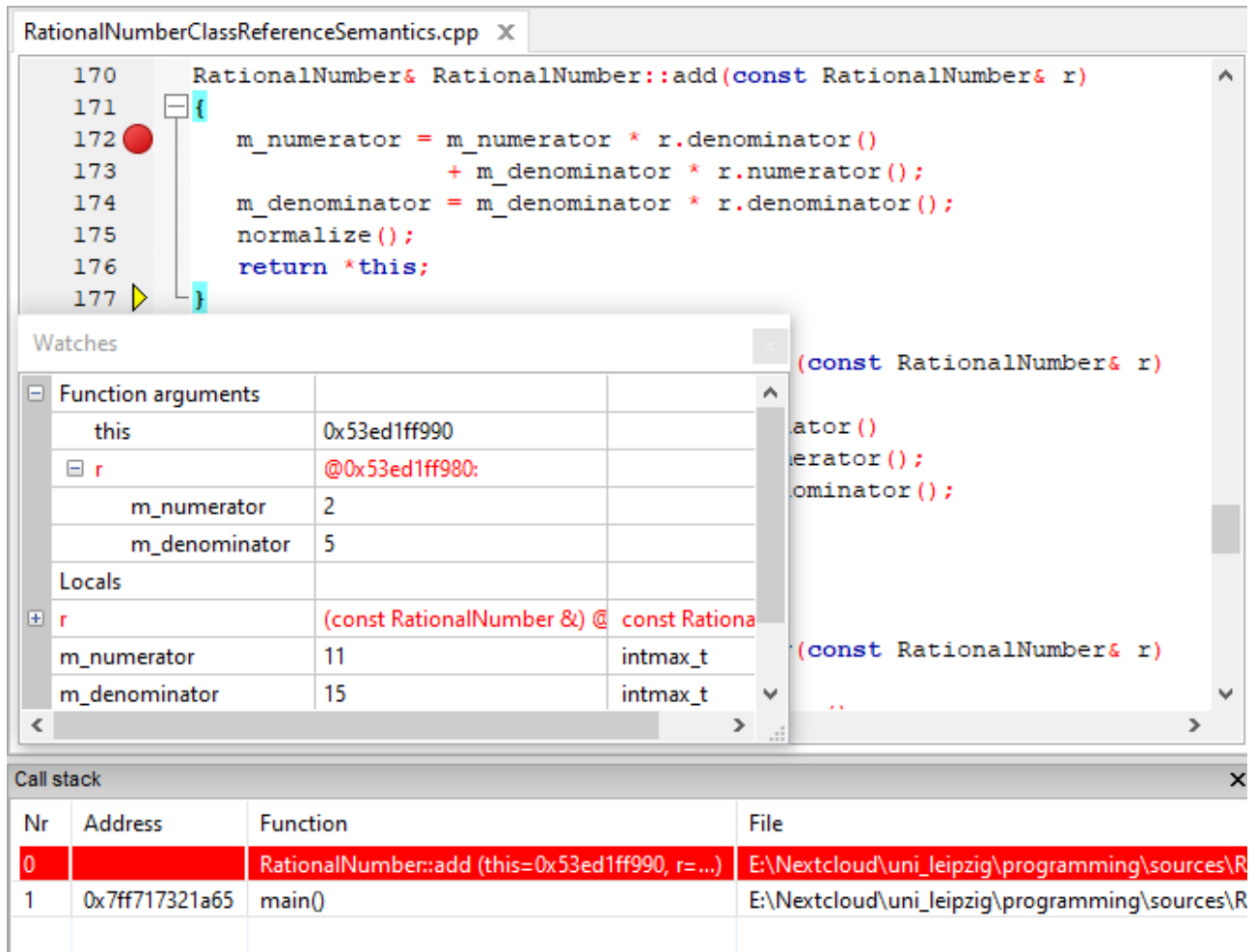


Figure 34: Debugging the reference-based program, part 2

The local variables `m_numerator` and `m_denominator`, i.e. the data members of the active object to which `this` points, now have the corresponding values.

With the help of the debugger it is possible to follow the execution of the program. The different debugging windows allow to switch interactively between the active function and its callers. In this way the debugger supports the understanding of the execution of the program in its different states. Exactly how this is done is difficult to explain in a written text. It should be explored in an interactive debugging session.

Next, the value-based design program is debugged (Figure 35). The arrangement of the data and the windows is exactly the same as before.

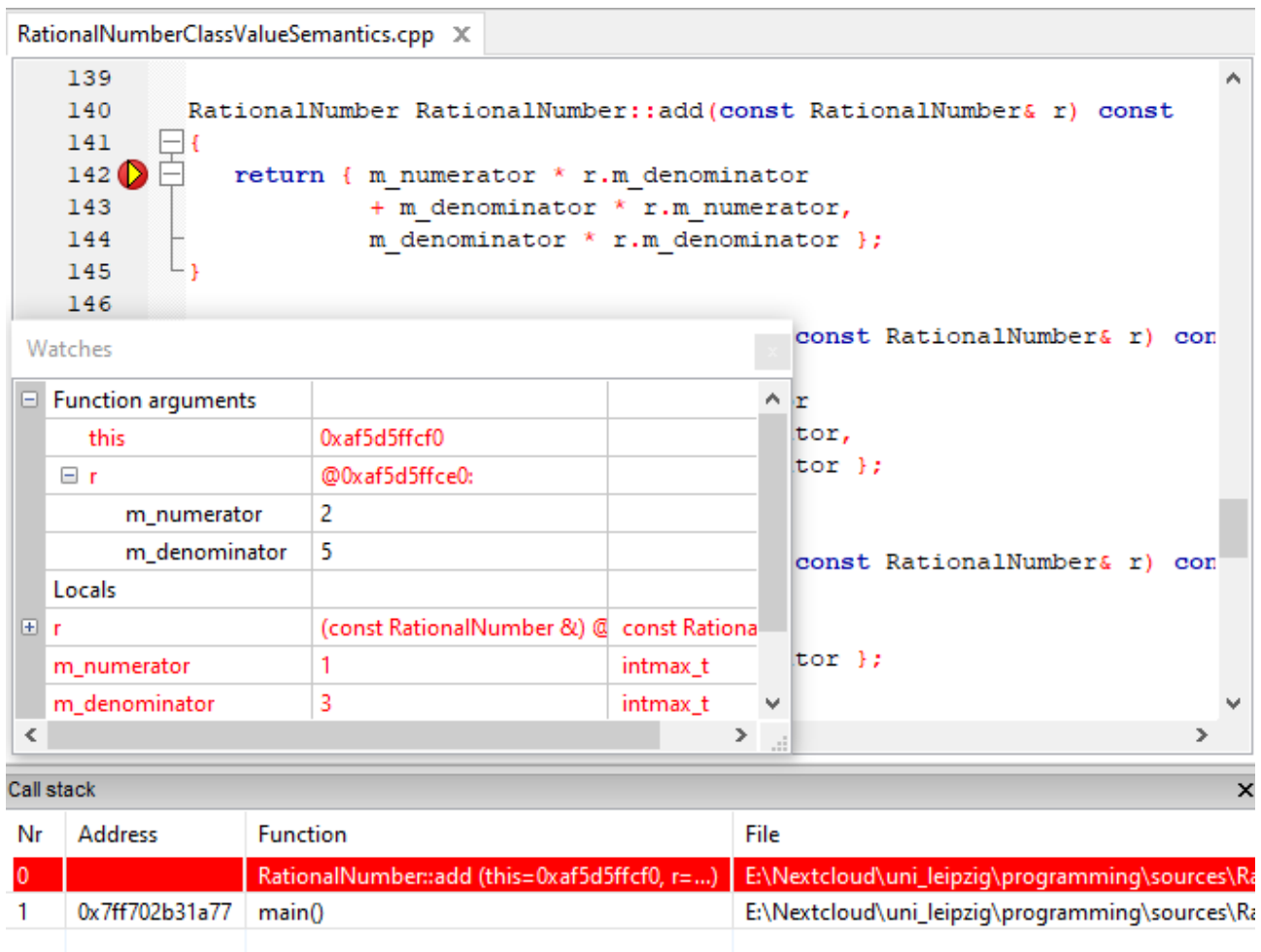


Figure 35: Debugging the value-based program, part 1

The calculation of the resulting rational number is spread over three lines. Of course, it is possible to step over the individual lines. But this does not change the content of the Watches window. This is because there is no variable for the temporary object created for the result in the Watches window, and the other variables do not change. If the debugger supports it, one can evaluate expressions or sub-expressions. The screenshot immediately before exiting `RationalNumber::add()` (Figure 36) shows that the watches and the function call stack have not been changed. As in the reference-based design version, it is of course possible to use the debugger to check whether variables have the expected values and in what order the functions call each other. Ultimately, this will not lead to surprises, since in most cases the objects in a value-based design do not have state changes during execution of the program.

Figure 36: Debugging the value-based program, part 2

RationalNumberClassValueSemantics.cpp X

```

139
140 RationalNumber RationalNumber::add(const RationalNumber& r) const
141 {
142     return { m_numerator * r.m_denominator
143             + m_denominator * r.m_numerator,
144             m_denominator * r.m_denominator };
145 }
146

```

Watches

Function arguments		
this	0xaf5d5ffc0	
r	@0xaf5d5ffce0:	
m_numerator	2	
m_denominator	5	
Locals		
r	(const RationalNumber &) @	const Rational
m_numerator	1	intmax_t
m_denominator	3	intmax_t

Call stack

Nr	Address	Function	File
0		RationalNumber::add (this=0xaf5d5ffc0, r=...)	E:\Nextcloud\uni_leipzig\programming\sources\R
1	0x7ff702b31a77	main()	E:\Nextcloud\uni_leipzig\programming\sources\R

5.4. Splitting of Programs

The program shown in Listing 85 consists of more than 200 lines of code, including comments and documentation comments. That's big enough to think about how to break it into smaller pieces. Why is this important? There are two main reasons:

1. The bigger a program is, the more time it takes to read and understand it.
2. Some parts of a program are very specific, but other parts can possibly be reused in other programs. Therefore, it is advantageous to effectively isolate these parts of a program.

Which guidelines help to identify suitable parts? *Modularity is an important principle of software engineering that can be applied here. Modularity is about breaking a program down into parts, called modules. Each module should have a high degree of cohesion. That is, it should consist of functions and types that are closely related, either because they depend on each other or because they belong together from a technical point of view. In addition, a module should have a low coupling. That means, it should have minimal dependencies on other modules.*

With modularity in mind, a first division can be made between `main()` and the rest of the program. The sole purpose of `main()` is to demonstrate the use of the `RationalNumber` class and its member functions, and free functions required to implement some of the member functions of `RationalNumber`.

The `sign()` function is used exclusively in `RationalNumber::normalize()`. This is `RationalNumber`'s only dependency on it. This function can be classified as a potentially useful general mathematical function that for some reason is not present in the C++ standard library. From a functional point of view, it can be classified as *math-helper function*. The rest is the `RationalNumber` class. Table 16 shows the corresponding classification of the functions of the `RationalNumberClassReferenceSemantics.cpp` program.

Name	main	rational_number	math_helper
Contains	<code>main()</code>	<code>RationalNumber</code>	<code>sign()</code>
Depends on	<code>RationalNumber</code> <code>sign()</code>	<code>sign()</code>	–

Table 16: Classification of functions

In the first line the previously identified modules are mentioned. The names *rational_number* and *math_helper* were chosen this way because they will be used as file names later. The second line indicates which elements are contained in the module listed in the table header. A look at the contained elements allows to judge the cohesion of the respective module. The third line indicates which elements in other modules the respective module depends on. This allows an evaluation of the coupling of the corresponding module.

Dependencies on parts of the standard library are not listed in Table 16, for example `<iostream>` or `<cmath>`.

Modularity is the prerequisite for another principle of software engineering, namely the already mentioned *information hiding (Abstract Data Types)*. This principle leads also to an additional division of the program files. To support the repeated use of modules such as *rational_number* and *math_helper*, these are further divided into *header* and *implementation files*. A header file contains all the information required to use its elements, and the corresponding implementation file contains all the code that is not relevant to the use of the header file. This division does not apply to *main*, since *main* is the client of `RationalNumber` and `sign()` and is not eligible for further use.

One convention – among others – is to append the extension *.hpp* to C++ header files, and the extension *.cpp* to implementation files. Of course, an implementation file must know the contents of its associated header file. Therefore, it includes its header file with the preprocessor directive `#include`. `math_helper.cpp` includes

math_helper.hpp, rational_number.cpp includes rational_number.hpp, and main.cpp includes rational_number.hpp to create exemplars of RationalNumber and use them.

Important Notice

To practically understand the following content, accessing a C++ compiler through a browser window is no longer sufficient. Instead, a C++ compiler and the necessary development tools, especially *make*, must be available for execution in a console window. The following examples use *g++* and *make*, which usually installs along with *g++*.

5.4.1. Header and Implementation Files

The steps for transforming a single program into modules that can be compiled separately and linked to form an executable program are introduced below. To highlight these steps, all documentation comments have been removed. They will be restored later.

In the following it is assumed that all files are contained in a (sub)directory named *RatNumRefSem_1*.

Listing 94 contains the prototype of the *math_helper* function. The function prototype has been slightly modified compared to the program in Listing 66. Since `intmax_t` is defined in `<stdint>`, this header file must be included. Since Listing 94 will be developed later in its own header file, a global or selective import of namespaces is not useful. Therefore, `intmax_t` must be fully qualified with its defining namespace, namely `std::intmax_t`.

```
1 #include <stdint> // Because of intmax_t
2
3 [[nodiscard]] std::intmax_t sign(const std::intmax_t& n);
```

Listing 94: *RatNumRefSem_1/math_helper.hpp*

The implementations of the `sign()` function is almost unchanged (Listing 95). Nevertheless, some details need to be explained.

```
1. #include "math_helper.hpp"
2.
3. using namespace std;
4.
5. intmax_t sign(const intmax_t& n)
6. {
7.     if (n < 0)
8.     {
9.         return -1;
10.    }
11.    if (n > 0)
12.    {
13.        return +1;
14.    }
```

```
15. return 0;
16. }
```

Listing 95: `RatNumRefSem_1/math_helper.cpp`

The corresponding header file containing the function prototype is included with `#include "math_helper.hpp"`. This differs from including header files from the standard library, for which angle brackets are used. The compiler, or more precisely the preprocessor, looks in the active directory for a header file enclosed in double quotes. If a header file is located in another directory, the name of the header file must be prefixed with either an absolute path or a relative path with respect to the active directory.

Since Listing 95 shows an implementation file, it is acceptable to import all namespace `std` identifiers at once as long as no collision of identifiers occurs. This is a good opportunity to provide some explanation about importing namespaces.

Namespaces or namespace identifiers should never be imported in a header file. Such an import would affect all files including this header file. It could lead to name collisions and would contradict the purpose of namespaces. **In header files, the identifiers of all namespace must be fully qualified.** In implementation files, it is a matter of personal taste or code writing guidelines whether to import identifiers from namespaces or to use fully qualified identifiers. Of course, if names of identifiers collide, the use of qualified identifiers is essential.

The header file that defines the `RationalNumber` class is shown in Listing 96. While this is a definition of `RationalNumber`, it does not contain the definitions of the member functions. These are contained in the corresponding implementation file.

```
1 #include <cstdint> // Because of intmax_t.
2
3 class RationalNumber
4 {
5     public:
6         RationalNumber(const std::intmax_t& n = 0, const std::intmax_t& d = 1);
7         [[nodiscard]] const std::intmax_t& numerator() const
8         {
9             return m_numerator;
10        }
11        [[nodiscard]] const std::intmax_t& denominator() const
12        {
13            return m_denominator;
14        }
15        RationalNumber& numerator(const std::intmax_t& n);
16        RationalNumber& denominator(const std::intmax_t& d);
17        RationalNumber& add(RationalNumber r);
18        RationalNumber& subtract(RationalNumber r);
19        RationalNumber& multiply(RationalNumber r);
20        RationalNumber& divide(RationalNumber r);
21        RationalNumber& input();
22        RationalNumber& output();
23    private:
24        void normalize();
25        std::intmax_t m_numerator { 0 },
26                    m_denominator { 1 };
27};
```

Listing 96: `RatNumRefSem_1/rational_number.hpp`

There are some changes compared to the program shown in Listing 85. The implementation of the constructor has been moved to the implementation file. This may result in a minor performance penalty that is unlikely to have any consequences for applications that use `RationalNumber`. Consequently, the implementation of `RationalNumber::normalize()` remains in the implementation file. In this way, `rational_number.hpp` is largely decoupled, since it includes only one header file. The implementations of the setter methods for `RationalNumber::m_numerator` and `RationalNumber::m_denominator` have also been shifted to the implementation file.

The most important change is that the arguments of arithmetic member functions are now passed by value rather than references to `const`. This eliminates the error when a `RationalNumber` is divided by itself. A copied argument object can never be the same as the receiver object of a member-function call. But this has consequences. When passing an argument as a reference to `const`, both a variable of the expected type and an arbitrary expression can be passed, for which a temporary object of the expected type can be automatically created. Passing an argument by value always creates a local variable of the expected type, either by copying or by applying a valid conversion. Also, passing an argument by value prevents *inclusion polymorphism*. What inclusion polymorphism is and the consequences of preventing it are not explained in this text.

Listing 96 shows only declarations of member functions of `RationalNumber`, in addition to two getter methods. The only changes from the previous versions are that the arguments of the arithmetic member functions are now passed by value so that they match their declarations in the corresponding header file.

Listing 97 shows the implementation file that corresponds to the header file shown in Listing 9. At the beginning, `<iostream>` is included as usual, and all the identifiers of namespace `std` are imported. This is fine in an implementation file, since it does not affect other files. Then `math_helper.hpp` and `rational_number.hpp` are included using double quotes, namely `#include "math_helper.hpp"` and `#include "rational_number.hpp"`.

```
1 #include <iostream>
2 #include <numeric> // Because of gcd().
3 using namespace std;
4
5 #include "math_helper.hpp"
6 #include "rational_number.hpp"
7
8 RationalNumber::RationalNumber(const intmax_t& n, const intmax_t& d):
9     m_numerator { n }, m_denominator { d }
10 {
11     normalize();
12 }
13
14 RationalNumber& RationalNumber::numerator(const intmax_t& n)
```



```

15 {
16     m_numerator = n;
17     normalize();
18     return *this;
19 }
20
21 RationalNumber& RationalNumber::denominator(const intmax_t& d)
22 {
23     m_denominator = d;
24     normalize();
25     return *this;
26 }
27
28 RationalNumber& RationalNumber::add(RationalNumber r)
29 {
30     m_numerator = m_numerator * r.denominator()
31                 + m_denominator * r.numerator();
32     m_denominator = m_denominator * r.denominator();
33     normalize();
34     return *this;
35 }
36
37 RationalNumber& RationalNumber::subtract(RationalNumber r)
38 {
39     m_numerator = m_numerator * r.denominator()
40                 - m_denominator * r.numerator();
41     m_denominator = m_denominator * r.denominator();
42     normalize();
43     return *this;
44 }
45
46 RationalNumber& RationalNumber::multiply(RationalNumber r)
47 {
48     m_numerator = m_numerator * r.numerator();
49     m_denominator = m_denominator * r.denominator();
50     normalize();
51     return *this;
52 }
53
54 RationalNumber& RationalNumber::divide(RationalNumber r)
55 {
56     m_numerator = m_numerator * r.denominator();
57     m_denominator = m_denominator * r.numerator();
58     normalize();
59     return *this;
60 }
61
62 RationalNumber& RationalNumber::input()
63 {
64     cout << "numerator: " << flush;
65     cin >> m_numerator;
66     do
67     {
68         cout << "denominator: " << flush;
69         cin >> m_denominator;
70         if (m_denominator == 0)
71         {
72             cerr << "Error, denominator may not be 0!"
73                 << endl;
74         }
75     } while (m_denominator == 0);
76     return *this;
77 }
78
79 RationalNumber& RationalNumber::output()
80 {
81     cout << '('

```

```

82     << m_numerator
83     << '/'
84     << m_denominator
85     << ')'
86     << flush;
87     return *this;
88 }
89
90 void RationalNumber::normalize()
91 {
92     intmax_t divisor = gcd(m_numerator,m_denominator);
93     m_numerator = sign(m_numerator) * sign(m_denominator)
94                 * abs(m_numerator) / divisor;
95     m_denominator = abs(m_denominator) / divisor;
96 }

```

Listing 97: `RatNumRefSem_1/rational_number.cpp`

Now it is possible to compile the implementation files separately. How this can be done is explained using a terminal window and `g++`. First, a new terminal window is opened in the `RatNumRefSem_1` directory on a Unix-based system. Then the dialog shown in Figure 37 is executed. User inputs are formatted in **bold**, corresponding system outputs are formatted in *italics*.

```

Last login: Mon Jun 7 17:09:17 on ttys000
user@computer RatNumRefSem_1 % ls
main.cpp          math_helper.hpp  rational_number.hpp
math_helper.cpp   rational_number.cpp
user@computer RatNumRefSem_1 % g++ -c -std=c++20 math_helper.cpp
user@computer RatNumRefSem_1 % ls
main.cpp          math_helper.hpp  rational_number.cpp
math_helper.cpp   math_helper.o    rational_number.hpp
user@computer RatNumRefSem_1 % g++ -c -std=c++20 rational_number.cpp
user@computer RatNumRefSem_1 % ls
main.cpp          math_helper.hpp  rational_number.cpp  rational_number.o
math_helper.cpp   math_helper.o    rational_number.hpp
user@computer RatNumRefSem_1 % g++ -c -std=c++20 main.cpp
user@computer RatNumRefSem_1 % ls
main.cpp          math_helper.cpp  math_helper.o        rational_number.hpp
main.o           math_helper.hpp  rational_number.cpp  rational_number.o
user@computer RatNumRefSem_1 % g++ -o demo math_helper.o rational_number.o main.o
user@computer RatNumRefSem_1 % ./demo
Enter 1st rational number
numerator: 1
denominator: 2
Enter 2nd rational number
numerator: 3
denominator: 4
sum = (5/4)
difference = (-1/4)
product = (3/8)
quotient = (2/3)
user@computer RatNumRefSem_1 %

```

Figure 37: `Separate compilation, linking and execution`

The `ls` command lists all files in a directory. Initially, the current directory contains only three `.cpp`- and two `.hpp`-files. Next, the compiler is executed, here `g++`. The `-c` option tells `g++` to compile only, without linking. The `-std=c++20` option enables the support for `C++20`, and the only file to compile is `math_helper.cpp`. Running `ls` again shows that `math_helper.o` was created by the previous execution of the compiler. The next command compiles `rational_number.cpp`. As the subsequent execution of `ls` shows, `rational_number.o` was created in this process. Now, `main.cpp` is compiled to `main.o`. The extension `.o` is the standard for object files. Finally, the compiler is invoked with the `-o` option. This option tells the compiler to create an executable file with the name specified next, here `demo`. Since the following arguments all refer to object files, the compiler performs only a link step to generate the specified output file. Without checking the contents of the directory again, `demo` is executed by typing `./demo`. The rest is the dialog with the `demo` program that calls the `main()` function.

5.4.2. Include Guard

Despite the fact that all implementation files have been compiled separately and linked into an executable application, more needs to be done. If a header file is included more than once, this can lead to problems due to duplicate definitions. Also mutual recursive inclusion of header files can occur in principle. For this reason, each header file must be prepared against multiple inclusion by using a so-called *include guard*. Listings 98 and 99 demonstrate include guards. It is assumed that the source files shown in Listings 98 and 99 are located in the subdirectory called `RatNumRefSem_2`.

```
1 #ifndef MATH_HELPER_HPP
2 #define MATH_HELPER_HPP
3
4 #include <stdint> // Because of intmax_t
5
6 [[nodiscard]] std::intmax_t sign(const std::intmax_t& n);
7 #endif // MATH_HELPER_HPP
```

Listing 98: `RatNumRefSem_2/math_helper.hpp`

The `#ifndef` preprocessor directive – its name is a contraction of *if not defined* – realizes *conditional inclusion*. If the following macro name is *not* defined, the following lines will be further processed by the preprocessor until the preprocessor reaches the corresponding `#endif` preprocessor directive. The next preprocessor directive, `#define`, defines exactly the macro name for which the preceding `#ifndef` checks whether it is defined. When this header file is included again, the macro name is already defined. Therefore, the preprocessor discards all following lines until `#endif` is reached.

```
1 #ifndef RATIONAL_NUMBER_HPP
2 #define RATIONAL_NUMBER_HPP
3
```

```

4 #include <stdint> // Because of intmax_t
5
6 class RationalNumber
7 {
8     public:
9         RationalNumber(const std::intmax_t& n = 0, const std::intmax_t& d = 1);
10        [[nodiscard]] const std::intmax_t& numerator() const
11        {
12            return m_numerator;
13        }
14        [[nodiscard]] const std::intmax_t& denominator() const
15        {
16            return m_denominator;
17        }
18        RationalNumber& numerator(const std::intmax_t& n);
19        RationalNumber& denominator(const std::intmax_t& d);
20        RationalNumber& add(RationalNumber r);
21        RationalNumber& subtract(RationalNumber r);
22        RationalNumber& multiply(RationalNumber r);
23        RationalNumber& divide(RationalNumber r);
24        RationalNumber& input();
25        RationalNumber& output();
26    private:
27        void normalize();
28        std::intmax_t m_numerator { 0 },
29                    m_denominator { 1 };
30 };
31 #endif // RATIONAL_NUMBER_HPP

```

Listing 99: *RatNumRefSem_2/rational_number.hpp*

It is a convention that macro names are fully capitalized. Usually, the macro name is the same as the header file name, with the dot replaced by an underscore. For example, `MATH_HELPER_HPP` is used as the macro name for the include file *math_helper.hpp*. It is a good idea to append the macro name as a comment to the corresponding `#endif` directive. This clarifies to which conditional include directive the `#endif` belongs.

These are the only changes to the files contained in the *RatNumRefSem_2* subdirectory. It is still possible to manually compile all implementation files and link them to an executable file, as shown above.

5.4.3. Preventing Name Collisions

In the header files all identifiers are defined in the global namespace. Therefore, when including other header files, name collisions may occur. To avoid this problem, identifiers in header files are placed in their own namespaces. Listing 100 shows how this is done. It is assumed that the source files shown in the following listings are located in sub directory *RatNumRefSem_3*.

```

1 #ifndef MATH_HELPER_HPP
2 #define MATH_HELPER_HPP
3
4 #include <stdint> // Because of intmax_t
5
6 namespace math_helper
7 {

```

```

8     [[nodiscard]] std::intmax_t sign(const std::intmax_t& n);
9 }
10 #endif // MATH_HELPER_HPP

```

Listing 100: *RatNumRefSem_3/math_helper.hpp*

The prototype of the `sign()` function is enclosed in the curly braces of the namespace defined by `namespace math_helper { /* ... */ }`. Everything declared here is a member of this namespace.

The implementations of the functions in the implementation file must be placed in the same namespace as Listing 101 shows. Alternatively, each function definition could be prefixed with the name of the declaring namespace followed by the scope operator, `::`. However, this leads to less change-friendly code and is therefore not shown in this text.

```

1 #include "math_helper.hpp"
2
3 namespace math_helper
4 {
5     using namespace std;
6
7     intmax_t sign(const intmax_t& n)
8     {
9         if (n < 0)
10        {
11            return -1;
12        }
13        if (n > 0)
14        {
15            return +1;
16        }
17        return 0;
18    }
19 }

```

Listing 101: *RatNumRefSem_3/math_helper.cpp*

This applies accordingly to *rational_helper*, as Listings 102 and 103 show. Both listings have been shortened to focus on the corresponding changes.

```

1 #ifndef RATIONAL_NUMBER_HPP
2 #define RATIONAL_NUMBER_HPP
3
4 #include <stdint> // Because of intmax_t
5
6 namespace rational_number
7 {
8     class RationalNumber
9     {
10    public:
11        /* ... */
12    private:
13        /* ... */
14    };
15 }
16 #endif // RATIONAL_NUMBER_HPP

```

Listing 102: *RatNumRefSem_3/rational_number.hpp (abridged)*

```

1 #include <iostream>
2 #include <numeric> // Because of gcd().

```

```

3 using namespace std;
4
5 #include "math_helper.hpp"
6 using namespace math_helper;
7
8 #include "rational_number.hpp"
9
10 namespace rational_number
11 {
12     /* ... */
13 }

```

Listing 103: *RatNumRefSem_3/rational_number.cpp (excerpt)*

In the implementation file shown in Listing 103, it is possible to import all identifiers of the `math_helper` namespace with `using namespace math_helper;`, because no name collisions occur. The file containing the `main()` function needs only a minor adjustment, as Listing 104 shows.

```

1 #include <iostream>
2 using namespace std;
3
4 #include "rational_number.hpp"
5 using namespace rational_number;
6
7 int main()
8 {
9     /* ... */
10 }

```

Listing 104: *RatNumRefSem_3/main.cpp (excerpt)*

After including `rational_number.hpp`, `using namespace rational_number;` was inserted. All these files together form a *software development project*, because they are the basis for the creation of the executable application. Due to the existing division of the files (header and implementation files) and the chosen namespaces, the overall coupling between these files is reduced to a necessary minimum.

Of course, it is still possible to compile the implementation files individually and link the object files manually to an executable file. However, this is not practical for larger projects and prone to errors. Therefore, the next step will be to automate this creation process.

5.4.4. Make

In practice, applications can consist of dozens or hundreds of source files with various dependencies and the project's documentation. Sometimes all intermediate files and automatically generated files need to be removed to put the project in a clean state. Other tasks may include testing and profiling with associated special artifacts. The following shows how the build process can be automated in terms of generating code and documentation and resetting the project using the classical tool *make*. Testing and profiling are not covered.

Figure 38 shows a so-called *dependency graph*, which displays the dependencies between the files of the project.

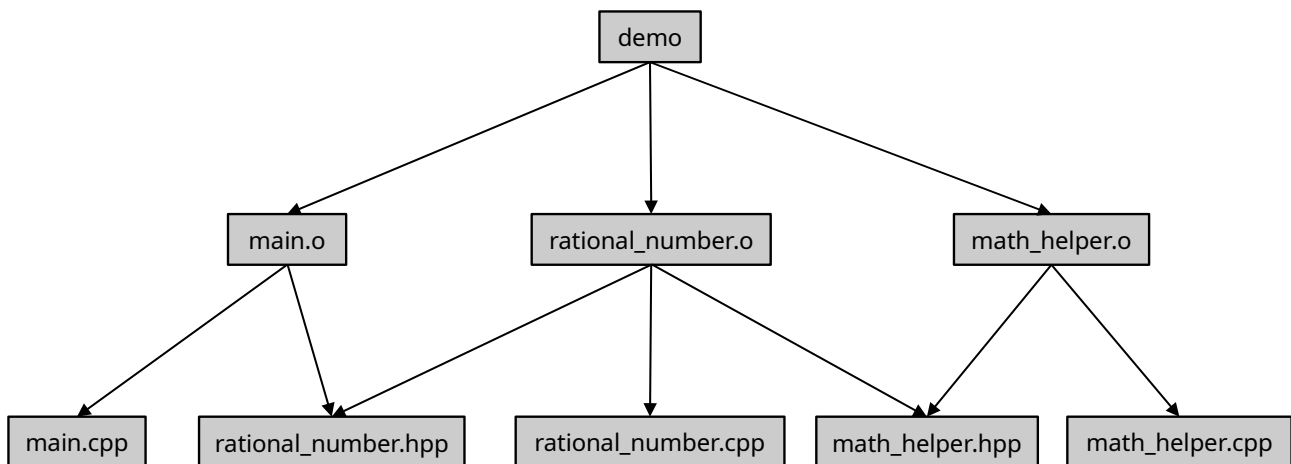


Figure 38: *Dependency graph*

The diagram is read from top to bottom. The boxes stand for files, the arrows for directed dependencies. The executable *demo* depends on the object files *main.o*, *rational_number.o* and *math_helper.o*. If any of the object files that *demo* depends on has a newer date than *demo*, *demo* must be linked again. The same is true if *demo* does not exist. If one of these object files does not exist, *make* looks for a rule to create it and executes that rule.

The object file *main.o* depends on the source files *main.cpp* and *rational_number.hpp*. If one of these source files is more recent than *main.o*, *main.o* must be recompiled. The same applies if *main.o* does not exist. Without looking at the source files, it is impossible to know that *main.cpp* includes *rational_number.hpp* and thus depends on this header file. Therefore, this dependency must be explicitly specified. This applies only to direct dependencies, but not to indirect ones. For example, it does not make sense to explicitly specify a dependency from *demo* to *rational_number.hpp*. Another dependency that must be explicitly specified is that from *rational_number.o* to *math_helper.hpp*, since *rational_number.cpp* includes *math_helper.hpp*.

The rest of the diagram can be read accordingly and requires no further explanation.

So-called *build automation tools* – *build tools* for short – rely on these dependencies to automate the build process of a piece of software including some of the associated artifacts. *make* is one of the oldest and most widely used build tools. It is usually installed together with *g++*. For this reason, in the following explanations of automating the build process, *make* is used for illustration. There are numerous

other build automation tools that cannot be covered here. Also *make* can only be explained superficially.

Having all the artifacts of a software project in a single directory would be confusing. Therefore, a software development project is organized in different directories. There is one directory that serves as the entry point for the project, for example, *RatNumRefSem*. Within this project directory, there are subdirectories that contain various files. In the following, it is assumed, that the *src* subdirectory contains the source files, the *obj* subdirectory contains object and executable files, and the *doc* subdirectory contains the documentation for the project. More specifically, the *src* subdirectory contains the *math_helper.hpp*, *math_helper.cpp*, *rational_number.hpp*, *rational_number.cpp*, and *main.cpp* files. The only difference from the corresponding files in the *RatNumRefSem_3* subdirectory is that all documentation comments have been restored. Figure 39 shows the overall directory structure.

```
RatNumRefSem/  
|  
+-----src/  
|  
+-----obj/  
|  
+-----doc/
```

Figure 39: Example directory structure for a C++ project

This directory structure can vary, of course.

To use *make*, the project should contain a simple text file named *GNUmakefile*, without any extension. This is similar to *Doxygen* looking for a file named *Doxyfile* in the active directory. If *make* cannot find *GNUmakefile* in the active directory, it looks for *makefile* or *Makefile* – in that order. With one of the options *-f file*, *--file=file*, or *-makefile=file*, *make* can be explicitly instructed to use *file* as makefile. For simplicity, a file named *makefile* is located in the project directory *RatNumRefSem*, so to build the project, a terminal window is opened in the *RatNumRefSem* directory and *make* is invoked without parameters to build the project.

Two basic components of a makefile are *comments* and *rules*. A comment starts with *#* and extends to the end of the line. A rule comprises one or more *targets*, *prerequisites*, and *recipes*. In most cases a target is the name of a file that needs to be created or updated. Usually there is only one target per rule. A target is followed by a colon. After the colon the prerequisites of the target are listed. Typically, these are the names of the files on which the target depends. *Depends on* means that the target must be created if it does not exist, or it is recreated if it is obsolete with respect to its prerequisites. If there is more than one prerequisite, they must be separated by spaces. The next line contains the recipe for (re)building the target. This recipe must be indented with exactly one tab. Using other whitespaces or more

than one tab is an error. A rule can have more than one recipe. All recipes must be indented with exactly one tab.

Both components are shown in Listing 105, which contains a simple makefile to create the executable file *demo* from the sources *main.cpp*, *rational_number.hpp*, *rational_number.cpp*, *math_helper.hpp*, and *math_helper.cpp*. All files and subdirectories should be located in a directory named *RatNumRefSem*.

```
1 # Makefile for RatNumRefSem, January 17, 2023, Author: Ulrich Eisenecker
2
3 # Rule 1: Link executable
4 obj/demo: obj/main.o obj/rational_number.o obj/math_helper.o
5     g++ -std=c++20 -o obj/demo obj/main.o obj/rational_number.o obj/math_helper.o
6
7 # Rule 2: Compile main.cpp
8 obj/main.o: src/main.cpp src/rational_number.hpp
9     g++ -std=c++20 -o obj/main.o -c src/main.cpp
10
11 # Rule 3: Compile math_helper.cpp
12 obj/math_helper.o: src/math_helper.cpp src/math_helper.hpp
13     g++ -std=c++20 -o obj/math_helper.o -c src/math_helper.cpp
14
15 # Rule 4: Compile rational_number.cpp
16 obj/rational_number.o: src/rational_number.cpp src/rational_number.hpp \
17     src/math_helper.hpp
18     g++ -std=c++20 -o obj/rational_number.o -c src/rational_number.cpp
19
20 # Rule 5: Delete all binaries and executable
21 clean:
22     rm -f obj/*
23     rm -rf doc/*
24
25 .PHONY: doc
26
27 # Rule 6: Generate documentation
28 doc:
29     doxygen
```

Listing 105: *RatNumRefSem/makefile.simple*

The first line contains a comment with some information. The following blank line is ignored as all blank lines. The third line contains a comment stating that rule 1 follows immediately.

The first rule is of particular importance. When a makefile is called without specifying a target, *make* attempts to update the target of the first rule. Other targets are updated only if necessary to update the goal. The term *goal* refers to the target which *make* should update.

The only target of rule 1 is *obj/demo*. If it does not exist or is obsolete with respect to one of the object files mentioned as a prerequisite, it is (re)built.

The recipe calls the compiler and tells it to link the object files to a file named *demo* in the subdirectory *obj*. The *-o* option specifies the name of the output file as next argument. The *-std=c++20* option tells the compiler to apply the latest C++ standard.

Actually, this is not relevant for linking, but it doesn't harm. For the recipes that compile source code, this option and its value are essential.

Rules 2 to 4 follow exactly the same scheme. Rule 4 introduces another new feature related to makefiles. Its first line ends with `\`. This symbol indicates that this line should continue with the next line as if it were written as a single line. There should be no characters after `\` other than *carriage return* or *line feed*.

Rule 5 is different. It uses a target that has no prerequisites, but two recipes. The recipes do not create a file named *clean*. Therefore, the recipes are always executed, when *make* is called with *clean* as parameter, namely *make clean*. The recipes execute the shell command `rm -f` to remove all files (and also subdirectories) contained in the *obj* and *doc* subdirectories. In this way, the entire project is put into a clean state with regard to all automatically created files.

Before rule 6, `.PHONY` tells *make* that the following dependencies are not files that need to be built. Therefore, specifying one of these as a targets will always execute the recipes associated with a corresponding target. Here, only *doc* is listed as a dependency. Running *make doc* in a terminal window will execute the corresponding recipe that calls *Doxygen* to generate the documentation. Since *Doxygen* is called without parameters, *Doxyfile* is used, which is also located in the project directory.

Of course, *clean* could also have been added as a prerequisite of `.PHONY`. This would prevent *make* from checking whether a file of that name exists, thus increasing the performance when executing the makefile. If a file named *clean* is created during the execution of the makefile, it must be listed as a dependency of `.PHONY`. Otherwise, its recipe(s) will not be executed once the file named *clean* is created.

When copying the makefile shown in Listing 105 by typing, care must be taken to ensure that each of the lines 5, 9, 13, 17, 21, 22, and 28 is indented by exactly one tab. When using copy & paste it must be ensured that the indentation of these lines is by exactly one tab and not by other whitespaces. Otherwise *make* cannot process the makefile correctly.

From a programming point of view, the makefile shown in Listing 105 has some shortcomings. If the chosen directory structure is changed, e.g. *./src* is renamed to *./source* for some reason, this change has to be done nine times! This situation is similar with the subdirectories *./obj* or *./doc*. Another example are the options to call the compiler or the compiler itself. Changing one of these requires several corrections in the makefile. This is annoying and error prone. This situation can be improved by using *variables*. The makefile in Listing 106 defines the variables *CXX*, *CXXFLAGS*, *SRCDIR*, and *OBJDIR* at the beginning. The variable names can be chosen arbitrarily. However, some variable names are also used by implicit rules, for

example, CXX and CXXFLAGS. Implicit rules are not explained here. Normally, variable names are completely capitalized.

```
1 # Makefile for RatNumRefSem, January 17, 2023, Author: Ulrich Eisenecker
2 # Version using variables
3
4 CXX = g++
5 CXXFLAGS = -std=c++20
6 SRCDIR = src
7 OBJDIR = obj
8
9 # Rule 1: Link executable
10 $(OBJDIR)/demo: $(OBJDIR)/main.o $(OBJDIR)/rational_number.o \
11 $(OBJDIR)/math_helper.o
12 $(CXX) $(CXXFLAGS) -o $(OBJDIR)/demo $(OBJDIR)/main.o \
13 $(OBJDIR)/rational_number.o $(OBJDIR)/math_helper.o
14
15 # Rule 2: Compile main.cpp
16 $(OBJDIR)/main.o: $(SRCDIR)/main.cpp $(SRCDIR)/rational_number.hpp
17 $(CXX) $(CXXFLAGS) -o $(OBJDIR)/main.o -c $(SRCDIR)/main.cpp
18
19 # Rule 3: Compile math_helper.cpp
20 $(OBJDIR)/math_helper.o: $(SRCDIR)/math_helper.cpp $(SRCDIR)/math_helper.hpp
21 $(CXX) $(CXXFLAGS) -o $(OBJDIR)/math_helper.o -c $(SRCDIR)/math_helper.cpp
22
23 # Rule 4: Compile rational_number.cpp
24 $(OBJDIR)/rational_number.o: $(SRCDIR)/rational_number.cpp \
25 $(SRCDIR)/rational_number.hpp $(SRCDIR)/math_helper.hpp
26 $(CXX) $(CXXFLAGS) -o $(OBJDIR)/rational_number.o -c \
27 $(SRCDIR)/rational_number.cpp
28
29 # Rule 5: Delete all binaries and executable
30 clean:
31 rm -f $(OBJDIR)/*
32 rm -rf doc/*
33
34 .PHONY: doc
35
36 # Rule 6: Generate documentation
37 doc:
38 doxygen
```

Listing 106: RatNumRefSem/makefile

A variable is defined by assigning a value to it with =, for example CXX = g++. To refer to this variable later, it must be surrounded by parentheses preceded by a dollar sign, for example, \$(CXX) -c math_helper.cpp, which becomes g++ -c math_helper.cpp. Now the compiler, options, or subdirectories can be easily changed, since there is exactly one point at the beginning of the makefile, where the corresponding variables are defined.

Lines 10, 12, 24, and 26 end with \, that is the continuation character introduced above.

There is much more to say about makefiles. For example, there are *implicit rules*, it is possible to use wildcards in filenames, makefiles can call other makefiles, and much more. Unfortunately, an adequate introduction to *make* can not be given here. A good source for more detailed information about *make* is the *make* manual itself, (*GNU Make Manual - GNU Project - Free Software Foundation*, n.d.). (Breymann,

2023) devotes a separate chapter to *make* and presents a project file for almost every C++ project, as the author himself claims. A compact and comprehensive introduction written in a casual tone is (Lambert, n.d.).

Finally, Listing 107 shows the configuration file for *Doxygen*. It has the same structure as the previously described configuration files for *Doxygen*.

```
1 # Doxyfile for RatNumberRefSem for Doxygen 1.9.1
2 PROJECT_NAME           = RatNumberRefSem
3 OUTPUT_DIRECTORY      = doc
4 INPUT                  = src/main.cpp src
5 GENERATE_LATEX         = NO
6 HAVE_DOT               = YES
7 CALL_GRAPH             = YES
8 CALLER_GRAPH           = YES
9 GRAPHICAL_HIERARCHY    = YES
10 DIRECTORY_GRAPH       = YES
```

Listing 107: *RatNumRefSem/Doxyfile*

While using *Doxygen version 1.9.1*, the author encountered a strange problem regarding the call graph generated for the `main()` function. Despite the fact that the member function `RationalNumber::divide()` is called on line 41, the call graph shows no box for `rational_number::RationalNumber::divide`. Converting lines 39 and 40 of `main()` to comments has the effect that *Doxygen* now creates a call graph for `main()` that includes the box for `rational_number::RationalNumber::divide`. Unfortunately, the author could not find the cause for this erroneous behavior.

5.5. Overloading Operators

In the next iteration, *operators* for mathematical operations and stream insertion and extraction for rational numbers will be added. In addition, the interface of `RationalNumber` will be slightly changed with respect to stream insertion and extraction.

Overloading operators does not provide new calculation capabilities. But the availability of operators for user-defined types can significantly increase the expressiveness and readability of programs.

5.5.1. Motivation for Overloading Operators

From the syntactic point of view, the mathematical formula $(a + b) / (a - c)$ is easy to understand. But how can this formula be represented with the abstract data type `RationalNumber` implemented with reference semantics (subdirectory *RatNumRefSem*)?

It is assumed, that `a`, `b` and `c` are variables of type `RationalNumber`.

The naive approach `(a.add(b)).divide(a.subtract(c))`; looks good. But it does not work as expected because `a` is modified by all arithmetic member functions of `RationalNumber`. The problem is illustrated by initializing `a` with $2/1$, `b` with $3/1$, and `c` with $5/1$.

The order of evaluating subexpressions by the `C++` compiler is not specified. In fact, the compiler can choose a different order when the same expression is evaluated again (*Order of Evaluation - Cppreference.Com*, n.d.). Therefore, both possibilities, i.e., evaluating `a.add(b)` first and then `a.subtract(b)`, as well as the reversed order, must be considered. Table 17 shows the first case, Table 18 the second.

Both possibilities lead to the same result, namely `a` being $1/1$, which does not correspond to the result of the evaluation of $(a + b)/(a - c)$ according to mathematical rules, namely $-5/2$.

State	Initial	<code>a.add(b)</code>	<code>a.subtract(c)</code>	<code>a.divide(a)</code>
Returned result	–	<code>a</code>	<code>a</code>	<code>a</code>
<code>a</code>	$2/1$	$5/1$	$1/1$	$1/1$
<code>b</code>	$3/1$	$3/1$	$3/1$	$3/1$
<code>c</code>	$4/1$	$4/1$	$4/1$	$4/1$

Table 17: Evaluation of `(a.add(b)).divide(a.subtract(c))`; (1st possibility)

State	Initial	<code>a.subtract(c)</code>	<code>a.add(b)</code>	<code>a.divide(a)</code>
Returned result	–	<code>a</code>	<code>a</code>	<code>a</code>
<code>a</code>	$2/1$	$-2/1$	$1/1$	$1/1$
<code>b</code>	$3/1$	$3/1$	$3/1$	$3/1$
<code>c</code>	$4/1$	$4/1$	$4/1$	$4/1$

Table 18: Evaluation of `(a.add(b)).divide(a.subtract(c))`; (2nd possibility)

The underlying problem is that calling a mathematical member function changes the receiver object in the reference-based design. One way to solve this problem is to introduce an additional variable as a copy of the initial value of the variable `a`, as shown in Listing 108.

```

1 RationalNumber a { 2,1 }, b { 3,1 }, c { 4,1 };
2 RationalNumber a_copy { a };
3 (a.add(b)).divide(a_copy.subtract(c));

```

Listing 108: Evaluation of `(a.add(b)).divide(a_copy.subtract(c))`;

Now, both possible evaluation sequences – they are shown in Tables 19 and 20 – give the expected result.

State	Initial	a.add(b)	a_copy.subtract(c)	a.divide(a_copy)
Returned result	–	a	a_copy	a
a	2/1	5/1	5/1	-5/2
a_copy	2/1	2/1	-2/1	-2/1
b	3/1	3/1	3/1	3/1
c	4/1	4/1	4/1	4/1

Table 19: Evaluation of (a.add(b)).divide(a_copy.subtract(c)) (1st possibility)

In both cases, a is now -5/2 which is the expected result.

In summary, the original mathematical formula must be transformed twice to implement it with RationalNumber in reference-based design, namely (a.add(b)).divide(a_copy.subtract(c)), where the declaration of a_copy is not shown.

State	Initial	a_copy.subtract(c)	a.add(b)	a.divide(a_copy)
Returned result	–	a_copy	a	a
a	2/1	2/1	5/1	-5/2
a_copy	2/1	-2/1	-2/1	-2/1
b	3/1	3/1	3/1	3/1
c	4/1	4/1	4/1	4/1

Table 20: Evaluation of (a.add(b)).divide(a_copy.subtract(c)) (2nd possibility)

The value-based design of RationalNumber requires only one transformation to represent the mathematical formula, namely RationalNumber result { a.add(b).divide(a.subtract(c)) }. This is because calling a mathematical member function in value-based design always returns a new object and does not change the receiver object. Therefore, the result of evaluating the expression is used to initialize the result variable. In value-based design, the result of a function call cannot be discarded (the corresponding member functions are declared with the [[nodiscard]] attribute). Instead, it must be used somehow, for example in another expression or to initialize another variable.

In both designs, overloading the arithmetic member functions of RationalNumber as operators allows the mathematical formula to be represented as (a + b)/(a - c). This is almost a one-to-one mapping from a requirement formulated in a mathematical notation to its programmed representation!

Operator overloading is an integral part of C++ and its standard library. Therefore, overloading the << and >> operators for stream insertion and extraction facilitates the input and output of RationalNumber exemplars from and to streams.

5.5.2. The Syntax of Operator Overloading

Basically, an operator is a function or a member function that can be called with an alternate syntax. This will be demonstrated later.

Many operators can be overloaded in C++. There are a few exceptions that cannot be overloaded, namely

- . – *Member access operator*, also known as *dot operator*
- .* – *Pointer to member operator*
- ? : – *Ternary or conditional operator*
- :: – *Scope resolution operator*
- sizeof – *Object size operator*
- typeid – *Object type operator*

Three of them, the member access operator, the scope resolution operator and sizeof, have already been used in the previous programs.

The remaining operators can be overloaded as member functions, and some of them as free functions as well. Listing 109 shows how to overload operator+() as a member function of the RationalNumber class in reference-based design.

```
1 RationalNumber RationalNumber::operator+(RationalNumber b) const
2 {
3     RationalNumber a { *this };
4     return a.add(b);
5 }
```

Listing 109: Overloading operator +() as a member function

An operator is declared with the keyword operator, followed by the symbol or the identifier of the operator, in this case +. This is followed by the parameter list. If the overloaded operator is *unary*, i.e. it has only one operand, the parameter list is empty when the operator is overloaded as a member function. In this case, the only operand is the receiver object, i.e. *this, which can be considered as implicit (first) operand. If a *binary operator* is overloaded as a member function, exactly one (explicit) parameter must be specified which serves as the second operand. All binary operators are *infix operators*, i.e. the first operand is to the left of the operator and the second operand is to the right of the operator. Therefore, all binary operators that are overloaded as member functions have exactly one explicit parameter. operator+() is a binary operator, so b is passed as a parameter of type RationalNumber. The only ternary operator in C++ is : ?, which cannot be

overloaded. Therefore, an overloaded operator can have at most two operands. Like all functions, the return type of an operator must be specified.

Type conversion operators have a somewhat different syntax; they are not explained here. If the execution of an operator must not change the receiver object, it should be declared as `const`, which is true in the concrete case.

A few more details need to be explained. First, the parameter `b` is passed by value to avoid the problem first reported in the [Class Design Based on Reference Semantics](#) Section. A second measure to address the problem is to first create a copy of the receiver object, `RationalNumber a { *this };`, before calling `RationalNumber::add()`. Also, the result is returned by value. Therefore, calling `RationalNumber::operator+()` changes `a`, but not the receiver object. Consequently, as mentioned earlier, the operator is marked as `const`. Since the addition of two rational numbers is already implemented in the method `RationalNumber::add()`, the corresponding code is not duplicated. Rather, the implementation of `RationalNumber::operator+()` calls this method.

With `a` and `b` as exemplars of `RationalNumber`, `operator+()` can be called with two syntax forms:

1. `a + b`
2. `a.operator+(b)`

The first form reflects the intention of operator overloading. The second form shows that an operator is indeed just a (member) function. Therefore, it can be called like a conventional (member) function.

There is an important aspect when overloading an operator as a member function: The first operand must be an exemplar of the type for which the operator is being overloaded. In this specific case, the receiver object must be a `RationalNumber`. This is exactly the implementation shown in Listing 109. For the second operand, i.e. the first parameter, this does not necessarily apply. Here, `b` is passed as a value. If `b` is a `RationalNumber`, a copy of `b` is created, and this copy is used within the operator. If `b` is not a `RationalNumber`, the compiler tries to convert `b` into a temporary object of type `RationalNumber`. The constructor of `RationalNumber` can be called with only one parameter. Therefore, this constructor becomes a *type conversion constructor*. Any integer value is sufficient to create an exemplar of `RationalNumber`. For example, `RationalNumber r { 2 };` produces a rational number whose `m_numerator` is 2 and whose `m_denominator` is 1. For this reason, `operator+()` can be called with any integer parameter as a second operand, for example `a + 2`. It must be mentioned that specifying a floating point value as a second operand, i.e. as a parameter, will cause a warning from the compiler since a floating point number is implicitly converted to an integer value.

Unfortunately, it cannot be called with an integer parameter as the first operand, e.g. `2 + a`. This is not possible because `operator+()` is overloaded as a member function, so the left operand *must* be a `RationalNumber`, as mentioned earlier.

This is where overloading `operator+()` as a free function comes into play, as shown in Listing 110.

```
1 RationalNumber operator+(RationalNumber a,RationalNumber b)
2 {
3     return a.add(b);
4 };
```

Listing 110: Overloading `operator+()` as a free function

In a free function, there is no receiver object that acts as the first operand. Therefore, all operands must be passed as parameters. Here the `RationalNumbers` `a` and `b` are passed by value. In this way, the problem of using an object as reference and as reference to `const` in the same function is basically avoided. Passing the first operand as a value has the added advantage that no further action is required since it is modified by the call to `a.add(b)`; in the implementation of `operator+()`. A free function cannot be declared as `const`, because there is no receiver object.

The compiler now performs automatic type conversion if at least one of the two operands is of type `RationalNumber`. If `a` and `b` are of type `RationalNumber`, all of the following expressions compile and execute correctly:

- `a + b`
- `a + 2`
- `2 + b`

Therefore, it is generally recommended here to implement the functionality of the binary operators as normal member functions of a type. The binary operators themselves are implemented separately as free functions that call the corresponding member functions of the respective type. This advice applies to all binary operators implemented for `RationalNumber`.

In addition, the `<<` and `>>` operators for stream insertion and extraction must be implemented as free functions, since their first operand is always a stream object. Implementing them as member functions of a stream type would require extending the source code of *all* stream classes, which is not acceptable for many reasons, including maintenance, testing, legal issues, and source code availability.

Some further details are to be mentioned. Not all operators can be overloaded as free functions. For example, type conversion operators – they are not explained in this text – can be implemented only as member functions. The syntax of operators cannot be changed. An overloaded operator has the same arity as its built-in counterpart, for example the division operator, `/`, always has two operands. Also, the precedence of an operator cannot be changed. For example, the assignment operator, `=`, always has a lower priority than arithmetic operators. Also, the

associativity of an operator cannot be changed. For example, the sequence operator, operator,(), (not explained here) is always evaluated from left to right (*left associativity*), while operator=() is always evaluated from right to left (*right associativity*). ***It is not possible to introduce new operators.***

In most cases it is possible to change the return type of operators. However, this may violate conventions and expectations. For example, if operator=() would return void, this would prevent assignment chaining, i.e. a = b = 42; would be no longer allowed. Therefore, operator=() should always return the receiver object of an assignment as a non-const reference. It is possible to change the semantics of an overloaded operator. This also violates conventions and expectations.

5.5.3. Overloading Operators for RationalNumber

As mentioned in the `Make` Section, the source code files in the `RatNumRefSem/src` subdirectory are not reproduced in this text. Nevertheless, some changes to the `rational_number.hpp` and `rational_number.cpp` files are presented below. They prepare the overloading of operator>>() and operator<<().

The member function `RationalNumber& RationalNumber::output()` sends its output to cout only. This must be changed so that it can send its output to any text stream. For this purpose, a parameter of type `std::ostream` is passed by reference. `std::ostream` serves as an interface for all types that are *output-text streams*. The signature of the modified member function is `void RationalNumber::output(std::ostream& os) const`. Since the modified member function returns nothing, i.e. void, it can be declared as const instead as a reference of type `RationalNumber` to `*this`. The stream object `os` passed as `std::ostream&` is modified by the output and therefore cannot be passed as a reference to const. One change in the implementation of `RationalNumber::output(std::ostream& os) const` is that `os` is used consistently instead of `cout`. If the output is sent to `cout`, it must simply be passed as a parameter, i.e. `a.output(cout);`. Another change is that `return *this;` is no longer required since the member function returns nothing.

The situation is different with `RationalNumber::input()`. This member functions interactively reads a rational number in a console window from the user. For this very purpose, this member function is preserved as it is.

The overloaded member function `RationalNumber::input(istream& is)` is the counter part of `RationalNumber::output(ostream& os)`. Its parameter `istream&` can be any *input text stream*. The member function reads all components of a rational number from this input text stream, but does no validity checking. As a consequence, any non-whitespace character can be used instead of (, /, and). Also, no check is made to see if the value read for `m_denominator` is not 0. So calling this method can put the receiver object in an invalid state! For this reason,

`RationalNumber::input(istream& is)` should not be called as `a.input(cin)`; to interactively input a `RationalNumber` from `cin`. Instead, `a.input()`; should be used.

The purpose of both member functions is *to serialize* a `RationalNumber` *into* a text stream, and *to deserialize* a `RationalNumber` *from* a text stream. Therefore both member functions should be implemented symmetrically. The serialization output of one function must be processed by the other function as input. Various errors related to streams can occur in both member functions. None of them are detected or handled. Errors related to streams and their handling will be discussed later.

The member function `RationalNumber::toLongDouble()` has been added for completeness and convenience. It returns a value of type `long double` which approximates the receiver object of type `RationalNumber` as floating point value. The implementation of this member function introduces a new operator, namely a *type cast operator*. The operator call `static_cast<long double>(m_numerator)` creates a new value of type `long double`, initialized with `m_numerator` of type `intmax_t`. It would have been sufficient to apply the type cast to only one of the data members of `RationalNumber`. This would trigger an automatic type conversion of the other component to perform floating point division. The type cast was explicitly applied to both components to highlight the necessary type conversions to perform a floating point division instead of an integer division. Of course, this member function is a candidate for implementation as a *type conversion operator*. The first reason not to so is that the existence of such a type conversion operator would conflict with the overloading of arithmetic operators as free functions. If both conversion options were available, the compiler would report an error because it could not decide which one to apply. The second reason is that type conversion operators are not covered in this text.

Listings 111 and 112 show only excerpts from the corresponding files, focusing on the previously introduced changes. The corresponding project is located in the `RatNumRefSemOp` subdirectory.

```
1 // ...
2 namespace rational_number
3 {
4 // ...
5 class RationalNumber
6 {
7     public:
8 // ...
9     void output(std::ostream& os) const;
10    void input(std::istream& is);
11    [[nodiscard]] long double toLongDouble() const;
12 // ...
13 };
14 }
```

Listing 111: `RatNumRefSemOp/rational_number.hpp` (excerpt)

```
1 // ...
2 namespace rational_number
```

```

3 {
4 // ...
5 void RationalNumber::output(ostream& os) const
6 {
7     os << '('
8         << m_numerator
9         << '/'
10        << m_denominator
11        << ')'
12        << flush;
13 }
14
15 void RationalNumber::input(istream& is)
16 {
17     char c;
18     is >> c
19         >> m_numerator
20         >> c
21         >> m_denominator
22         >> c;
23 }
24
25 long double RationalNumber::toLongDouble() const
26 {
27     return static_cast<long double>(m_numerator) /
28            static_cast<long double>(m_denominator);
29 }
30 }

```

Listing 112: *RatNumRefSemOp/rational_number.cpp (excerpt)*

The operators are kept separate from `RationalNumber`. This makes it possible to include them only when they are needed. However, they are also in the `rational_number` namespace. Listing 113 shows the header file and Listing 114 shows the implementation file.

```

1 /*! \file rational_number_operators.hpp
2 *
3 * Operators related to rational numbers as free functions.
4 *
5 * \author Ulrich Eisenecker
6 * \date June 21, 2021
7 */
8
9 /*!
10 * Include guard for rational_number_operators.hpp
11 */
12 #ifndef RATIONAL_NUMBER_OPERATORS_HPP
13 #define RATIONAL_NUMBER_OPERATORS_HPP
14
15 #include "rational_number.hpp"
16
17 /*! Namespace for types and functions related to rational numbers.
18 */
19 namespace rational_number
20 {
21     /*! Adds rational numbers a and b
22     * and returns result as value.
23     */
24     [[nodiscard]] RationalNumber operator+(RationalNumber a, RationalNumber b);
25     /*! Subtracts rational number a from b
26     * and returns result as value.
27     */
28     [[nodiscard]] RationalNumber operator-(RationalNumber a, RationalNumber b);
29     /*! Multiplies rational numbers a and b

```

```

30     * and returns result as value.
31     */
32     [[nodiscard]] RationalNumber operator*(RationalNumber a,RationalNumber b);
33     /*! Divides rational number b by a
34     * and returns result as value.
35     */
36     [[nodiscard]] RationalNumber operator/(RationalNumber a,RationalNumber b);
37
38     /*! Stream-insertion operator which outputs RationalNumber to output stream.
39     */
40     std::ostream& operator<<(std::ostream& os,const RationalNumber& r);
41     /*! Stream-extraction operator which inputs RationalNumber from input stream.
42     */
43     std::istream& operator>>(std::istream& is,RationalNumber& r);
44 }
45 #endif // RATIONAL_NUMBER_OPERATORS_HPP

```

Listing 113: *RatNumRefSemOp/rational_number_operators.hpp*

```

1 #include <iostream>
2 using namespace std;
3 #include <numeric> // Because of gcd().
4
5 #include "rational_number.hpp"
6 #include "rational_number_operators.hpp"
7
8 namespace rational_number
9 {
10     RationalNumber operator+(RationalNumber a,RationalNumber b)
11     {
12         return a.add(b);
13     }
14
15     RationalNumber operator-(RationalNumber a,RationalNumber b)
16     {
17         return a.subtract(b);
18     }
19
20     RationalNumber operator*(RationalNumber a,RationalNumber b)
21     {
22         return a.multiply(b);
23     }
24
25     RationalNumber operator/(RationalNumber a,RationalNumber b)
26     {
27         return a.divide(b);
28     }
29
30     std::ostream& operator<<(std::ostream& os,const RationalNumber& r)
31     {
32         r.output(os);
33         return os;
34     }
35
36     std::istream& operator>>(std::istream& is,RationalNumber& r)
37     {
38         r.input(is);
39         return is;
40     }
41 }

```

Listing 114: *RatNumRefSemOp/rational_number_operators.cpp*

All parameters of the mathematical operators are passed by value. This is consistent to the corresponding mathematical member functions and avoids problems with

unexpected changes of the passed parameters. Even the change of parameter passing in the corresponding mathematical member functions back to reference to `const` would be compensated this way. Consequently, the results are also returned by value. Each mathematical operator is declared with the `[[nodiscard]]` attribute, since ignoring its result would be nonsensical.

The latter is different for `operator<<()` and `operator>>()`. Both operators return the stream passed to them as a reference to allow chained input or output. Ultimately, the stream returned is not used for any other input or output and must therefore be ignored. Therefore, `[[nodiscard]]` cannot be used here. Also, these operators must accept and return references to streams, not references to `const` streams, since any input or output modifies the corresponding stream.

In this way, the mathematical and the stream operators fully comply with the earlier recommendation to implement the functionality of the operators as ordinary member functions and to overload the operators as free functions calling the corresponding member functions. Of course, this is only possible if these operators can be overloaded as free functions. Some authors and some implementations do not follow this recommendation. This regularly leads to the situation where operators overloaded as free functions require access to private data members of objects, which breaks encapsulation and information hiding. This should be considered bad style. How to break encapsulation and information hiding is not explained in this text.

The `main()` function (Listing 115) focuses on the use of the overloaded operators and the recently introduced member function `RationalNumber::toLongDouble()`. It also introduces so-called *string streams* for the first time. A string stream is a special stream object that resides in memory rather than on a mass storage. To use string streams the `<sstream>` header file must be included. The statement `ostringstream oss;` declares an *output string stream* called `oss` (short for *output string stream*). After the declaration, `oss` can be used like `cout`. Everything sent to `oss` is written to `oss`'s internal string buffer and does not appear anywhere else.

The contents of the output string stream can be accessed with the member function `ostringstream::str()`. This member function returns the current content of its buffer as a new object of type `std::string`. This string can be used like any other string. For example, it can be sent to `cout`. In `main()` it is used to initialize the *input string stream* declared by `istringstream iss(oss.str());`. In this way, the previously generated output becomes the input. Any input that can be read from `cin` can also be extracted from an input string stream, here `iss` (short for *input-string stream*). However, there is an important difference! An input string stream is not suitable for implementing interactive user input, as `RationalNumber::input()` does. This requires an additional output stream and an entity such as a user to respond to error messages and provide new and corrected input. With the

exception of cin (in combination with cout) this restriction applies to all other input streams.

These explanations should be enough to understand what main() does and how.

```
1  /*! \file main.cpp
2  *
3  *  Demo application for rational numbers
4  *  and corresponding operators
5  *
6  *  \author Ulrich Eisenecker
7  *  \date January 8, 2024 */
8
9  #include <iostream>
10 #include <sstream>
11 using namespace std;
12
13 #include "rational_number.hpp"
14 #include "rational_number_operators.hpp"
15 using namespace rational_number;
16
17 /*! Calls operators related to RationalNumber.
18 */
19 int main()
20 {
21     cout << "Please enter rational number"
22         << endl;
23     RationalNumber a { };
24     a.input();
25     RationalNumber b { 2,3 },
26                    c { 5,7 },
27                    d { 11,13 };
28     RationalNumber sum { a + b + c + d };
29     cout << a << " + " << b << " + "
30         << c << " + " << d << " = "
31         << sum
32         << endl;
33     cout << sum << " = "
34         << sum.toLongDouble()
35         << endl;
36     ostringstream oss;
37     oss << (a - b) << (b * c) << (c / d);
38     RationalNumber difference, product, quotient;
39     istringstream iss { oss.str() };
40     iss >> difference >> product >> quotient;
41     cout << a << " - " << b << " = "
42         << difference << endl
43         << b << " * " << c << " = "
44         << product << endl
45         << c << " / " << d << " = "
46         << quotient << endl;
47 }
```

Listing 115: *RatNumRefSemOp/main.cpp*

In the following it is assumed that the reader has converted the program *RationalNumberClassValueSemantics.cpp* (Listing 93) into a project with its own files and a makefile similar to the project *RatNumRefSem*. All necessary files should be located in the subdirectory *RatNumValSem*, whose structure is parallel to *RatNumRefSem*. This transformation must be complete, and each goal of the corresponding makefile must be executed without errors and return the expected

result. Based on this project, the changes to obtain *RatNumValSemOp* are presented below.

The general goal is to add the same operators as in *RatNumRefSemOp* to the value-based design of `RationalNumber`. The resulting project should be located in a subdirectory named *RatNumValSemOp*.

Ideally, the solution sought should be as close as possible to the reference-based design. This requires some changes to `RationalNumber` which are presented below.

The first change concerns the member function `RationalNumber::output()`, to which a reference to `ostream` named `os` is added as a parameter. The modified implementation uses `os` instead of `cout`.

Secondly, once a `RationalNumber` has been created in the value-based design, it should not be modified. This cannot be maintained if a `RationalNumber` is to be entered using `operator>>()`. Therefore, the member function called by `operator>>()` must set new values for an existing exemplar of `RationalNumber`. For this purpose, `RationalNumber::input(istream& is)` is introduced as a member function. It has the same implementation as the corresponding member function in the reference-based design. Therefore, the same restrictions and caveats apply. It should be noted that the presence of this member function violates the philosophy of value-based design. But in order to integrate a value-based design of `RationalNumber` into common C++ practices, this compromise must be made.

Third, the free function `inputRationalNumber()` is renamed to `input()` and moved to `RationalNumber` as a static member function. Of course, `inputRationalNumber()` is in the `rational_number` namespace and thus does not pollute the global namespace. This change is not strictly necessary, but harmonizes both designs and strengthens the cohesion of `RationalNumber`.

A static member function is declared with the keyword `static`, e.g. `static RationalNumber input()`. `static` must not be repeated in a separate definition of a member function. A static member function does not need an exemplar of its class to call it. For this reason, `this` (the pointer to the receiver object) is not defined in a static member function, since there is no receiver object. Also, a static member function cannot be declared as `const`, since there is no receiver object associated with it that may not be modified. A static member function can be thought of as a kind of free function that resides in the quasi-namespace defined by its class. To call it, it must be preceded with the name of its class and the scope operator, for example, `RationalNumber::input()`. Since the result of calling `RationalNumber::input()` should not be ignored, its declaration is preceded by the `[[nodiscard]]` attribute.

A fourth change is the introduction of the member function `RationalNumber::toLongDouble()` to bring it in line with the revised reference-based design.

Listings 116 and 117 only show the differences to the *RatNumValSem* project, which is assumed to have been created by the reader

```
1 // ...
2 namespace rational_number
3 {
4 // ...
5 class RationalNumber
6 {
7     public:
8 // ...
9     void output(std::ostream& os) const;
10    void input(std::istream& is);
11    [[nodiscard]] long double toLongDouble() const;
12    [[nodiscard]] static RationalNumber input();
13 // ...
14 };
15 }
16 #endif // RATIONAL_NUMBER_HPP
```

Listing 116: *RatNumValSemOp/rational_number.hpp*

```
1 // ...
2 namespace rational_number
3 {
4 // ...
5 void RationalNumber::output(ostream& os) const
6 {
7     os << '('
8         << m_numerator
9         << '/'
10        << m_denominator
11        << ')'
12        << flush;
13 }
14
15 void RationalNumber::input(istream& is)
16 {
17     char c;
18     is >> c
19         >> m_numerator
20         >> c
21         >> m_denominator
22         >> c;
23 }
24
25 long double RationalNumber::toLongDouble() const
26 {
27     return static_cast<long double>(m_numerator) /
28            static_cast<long double>(m_denominator);
29 }
30
31 RationalNumber RationalNumber::input()
32 {
33     intmax_t numerator { 0 },
34             denominator { 1 };
35     cout << "numerator: " << flush;
36     cin >> numerator;
37     do
38     {
39         cout << "denominator: " << flush;
40         cin >> denominator;
41         if (denominator == 0)
42         {
43             cerr << "Error, denominator may not be 0!"
44                  << endl;

```

```

45     }
46     } while (denominator == 0);
47     return RationalNumber { numerator, denominator };
48 }
49 // ...
50 }
51 }

```

Listing 117: RatNumValSemOp/rational_number.cpp

With respect to the adapted version of `RationalNumber` of the value-based design, the implementation of the desired operators differs only slightly from *RatNumRefSemOp/src/rational_number_operators.hpp* and the associated implementation file. Since there is no aliasing problem in the value-based design, all parameters of type `RationalNumber` are passed as a reference to `const`. Since this is the only exception, the implementation of the operators is identical.

Both implementations of `main()` differ only in two lines. In the reference-based design, `RationalNumber a` is initialized by the standard constructor. Then, `a.input()` is called to allow the user to enter new values. Listing 118 shows the corresponding snippet. Lines 1 and 2 in Listing 118 correspond to lines 23 and 24 in the original file.

```

1 RationalNumber a { };
2 a.input();

```

Listing 118: Declaring and inputting a RationalNumber (reference-based design)

In the value-based design, both lines are replaced by initializing `RationalNumber a` by calling the static member function `RationalNumber::input()` (Listing 119). Line 1 in Listing 119 corresponds to line 23 in the original file. The line ends with a comma because additional variables of type `RationalNumber` are declared in the following lines.

```

1 RationalNumber a { RationalNumber::input() },

```

Listing 119: Declaring and inputting a RationalNumber (value-based design)

It is noteworthy that both designs, reference-based and value-based, started out differently but became more similar as their functionality expanded. This was mainly due to practical needs, such as avoiding aliasing problems and integrating well with common C++ practice, rather than design philosophy.

5.6. Testing

Meanwhile, `RationalNumber` plus related functions and operators in both designs comprises about 20 functions and member functions. This is a good occasion to think again about testing. Many changes and extensions have been implemented, and there has always been a need to trust that the changes do not break existing

code or introduce bugs. Are there ways to test this more systematically and automatically? Fortunately, the answer is yes, as explained below.

5.6.1. Motivation for Automated Testing

Unit testing is a method of checking for errors once a unit has been changed. There are many frameworks for unit testing. And of course, unit testing can also become part of the build process. In this way, a unit test can be executed automatically when a source file is changed. In addition, the execution of the build tool can be triggered automatically by a higher-level process. This will not be explored in this text.

In programming, a unit is a piece of software with a well-defined interface and a (possibly hidden) implementation that can be used (more or less) separately. According to this definition, the smallest unit is a function. It has a well-defined interface, i.e. the prototype, and an implementation that need not be available to use the function. Moreover, the function can be called separately. Nevertheless, it may be necessary to especially prepare the parameters for calling the function. And of course, the function may call other functions to accomplish its task. ***A class can also be considered as a unit. An assembly of cooperating classes, such as a component or a framework, is not considered a unit because its parts are more or less loosely coupled and such assemblies can usually be extended with new parts.*** Therefore, unit tests are usually not suitable for testing such assemblies.

A unit test serves various purposes:

1. It checks whether a unit conforms to its specification. If this is the case, the test is successful.
2. It checks whether the unit runs correctly for different parameter values or combinations of parameter values. In practice, it is usually impossible to test all parameter values and all combinations of them.
3. It checks whether the unit still runs as expected after a change in its implementation. Sometimes previous errors are reintroduced with more recent changes to a unit. This phenomenon is called *regression of errors*. Therefore, all previous test cases should be preserved and re-executed whenever a change is made to a unit. This is called *regression testing*.

Because of possible interactions between the units it may be useful to perform an overall test of all units to check for known or new errors. This is called an *integration test*.

When a unit test fails, there are of course two possible places where the error originates. One place is the unit under test. The other place is the unit test itself. A unit test is a piece of software and as such is error prone like any software. The development of a unit test requires as much attention and effort as the development

of the software under test. It may even happen that the unit test comprises more lines of code than the software under test.

There are several methods for performing unit tests. One method is to develop software and implement and execute unit tests at an entirely different time. This method carries the risk that implementing unit tests may be perceived as a burden, and executing unit tests may reveal so many errors that it takes a lot of time and effort to detect and fix them. Of course, this is not the best method for unit testing. Nevertheless, it is the method used here for testing `RationalNumber` and its associated functions in both designs.

Another approach is to implement a unit test first and then the unit under test. This approach, called *test first*, is the central idea of *test-driven design*. This approach assumes that the requirements for the unit are precisely known in order to implement the unit test. In this way, the unit test can be considered as an instance of the formalized requirements of the unit.

Of course, there are many variations of the above methods. ***In general, it is advisable to write and run a unit test close to the time when the unit is developed. It is also advisable to start the unit tests with the basic units first and end with the tests of more complex units.*** In this way, the likelihood that more complex units will be built on well tested, simpler units increases. If a bug is discovered, the search in the higher-level units is likely to be more successful. If lower-level units have not yet been tested, the search for the cause often has to start at lower levels, while it cannot be ruled out that the error has its origin at a higher level.

As pointed out before, there are many frameworks for unit testing that also apply to C++. Very well known are the *Boost Test Library (Part IV. Boost Test Library: The Unit Test Framework, n.d.)* and *GoogleTest (Google/Googletest, 2015/2021)*. However, for testing `RationalNumber` and its associated operators *Catch2 (Catchorg/Catch2, 2010/2021)* is used as framework for unit testing. It is relatively lightweight and easy to use, for example, by simply including a single header file. Its terminology sometimes differs from other testing frameworks. In the following, *Catch2* is introduced as far as it is necessary for the understanding of the presented tests. Its special features and differences to other testing frameworks will not be discussed.

To use *Catch2*, it should be downloaded and unzipped to a suitable directory. Then the header file `catch.hpp` contained in the unzipped directory `Catch2-2.x/single_include/catch2` (or a corresponding directory in a newer version of *Catch2*) should be copied and pasted into the directory containing the source files of the project under test.

5.6.2. General Design of Tests

The units to be tested, i.e. the functions and a class, are declared in *math_helper.hpp*, *rational_number.hpp*, and *rational_number_operators.hpp*. A workable approach is to organize the tests around these header files. Each *.cpp* file with the *_test* suffix in its name tests the units of the corresponding *.hpp* file, e.g. *math_helper_test.cpp* tests the functions declared in *math_helper.hpp*.

These tests are compiled separately. In principle, they can also be executed as individual test applications. Since all tests are to be executed at once, there is a special file that serves as application for executing all tests. Therefore, this file is called *full_test.cpp*. Listing 120 shows its contents.

```
1 #define CATCH_CONFIG_MAIN
2 #include "catch.hpp"
3
4 #include "math_helper_test.cpp"
5 #include "rational_number_test.cpp"
6 #include "rational_number_operators_test.cpp"
```

Listing 120: *RatNumRefSemOp/src/full_test.cpp*

The first two lines must be present to use *Catch2*. The definition of the *CATCH_CONFIG_MAIN* macro causes *Catch2* to provide a *main()* function that executes the tests. The following *#include* directive includes the *catch.hpp* header file, which contains everything needed to write and run the tests. This must be done before including the files that contain the various tests. Otherwise the program will not compile because all the macros, types and functions required to write the test cases are not yet known.

5.6.3. Testing the Mathematical Helper Function

Listing 121 shows the tests for the *sign()* function declared in *RatNumRefSemOp/src/math_helper.hpp*.

```
1 // math_helper_test.cpp by Ulrich Eisenecker, February 8, 2022
2
3 #include <limits> // Because of numeric_limits<>.
4 #include <stdint> // Because of intmax_t.
5
6 #include "math_helper.hpp"
7
8 TEST_CASE("sign", "[math_helper]")
9 {
10     using namespace std;
11     using namespace math_helper;
12     // Testing smallest values.
13     REQUIRE(sign(-1) == -1);
14     REQUIRE(sign( 0) ==  0);
15     REQUIRE(sign( 1) ==  1);
16     // Testing largest values.
17     intmax_t minValue { numeric_limits<intmax_t>::min() },
18             maxValue { numeric_limits<intmax_t>::max() };
19     REQUIRE(sign(minValue) == -1);
```

```
20 REQUIRE(sign(maxValue) == 1);  
21 }
```

Listing 121: `RatNumRefSemOp/src/math_helper_test.cpp`

The `<limits>` header file is included because of `numeric_limits<>`, which is required for testing the `sign()` function. The next preprocessor directive includes the `math_helper.hpp` header file, which contains the prototypes of the functions to be tested. Then the first test case is defined with the macro `TEST_CASE`. This macro can take one or two arguments. The following examples always use two arguments.

The first argument is a string with the name of the test case. This name must be unique for all test cases, because it identifies the test case. Here the test case is called "sign". The optional second argument is also a string and contains one or more tags. Each tag must be enclosed in square brackets, i.e. "[...]". Here the only tag is "[math_helper]". How to use the name and the tags will be explained later.

The implementation of the test case is enclosed in curly braces { ... }. Ideally, the test cases are written in such a way that they are isolated from each other, i.e. **no test case should depend on another test case**. It should be possible to insert new test cases, delete old ones, or to move test cases within the same file or between files. For this reason, the namespaces `std` and `math_helper` are imported in each test case and not before.

There are various strategies for testing. For example, tests with special small and large parameter value pairs, and so-called *happy path tests*. A happy path test uses arbitrary values for which the test is normally successful.

So-called *assertion macros* (*Catch2* terminology) are used to formulate the conditions that must hold for a successful test. The assertion macro `REQUIRE()` takes a simple expression which is evaluated. The result of the evaluation is recorded. If all assertion macros evaluate to true, the test case passes. Otherwise, the violated assertion is reported including the value of the expression. `REQUIRE(sign(-1) == -1);` calls the `sign()` function with parameter `-1` and checks if the result is equal to `-1`. All following conditions are also formulated with `REQUIRE()`. Additional assertion macros are documented in (*Catchorg/Catch2*, 2010/2021). Assertion macros can only handle simple expressions. If the expression is too complex to be processed by the assertion macro, a variable must be initialized beforehand with the result of the evaluation of the complex expression. Then, this variable can be used to formulate a simpler expression that can be processed by the assertion macro.

The remaining tests of the `sign()` function use the special values `0`, and `1`. `0` is a special value because the sign of `0` is defined as `0`. In addition, the minimum and maximum values are tested, which are determined platform-independently with `numeric_limits<>`.

After all this has been described, the test case should be self-explanatory.

5.6.4. Testing of RationalNumber

Testing of RationalNumber is more extensive due to the larger number of member functions to be tested. Incidentally, only public member functions are tested, since private or protected member functions are not accessible from outside the class. Therefore, private or protected member functions can only be tested indirectly via extensive testing of public member functions, possibly with special parameter values to trigger their indirect execution. Listing 122 shows all test cases for RationalNumber.

```
1 // rational_number_test.cpp by Ulrich Eisenecker, June 26, 2021
2
3 #include <sstream>
4 #include "rational_number.hpp"
5
6 TEST_CASE("constructor", "[rational_number]")
7 {
8     using namespace rational_number;
9     // All constructor parameters are defaulted.
10    RationalNumber r1 { };
11    REQUIRE(r1.numerator() == 0);
12    REQUIRE(r1.denominator() == 1);
13    // Last constructor parameter is defaulted.
14    RationalNumber r2 { 1 };
15    REQUIRE(r2.numerator() == 1);
16    REQUIRE(r2.denominator() == 1);
17    // All constructor parameters are explicitly provided.
18    RationalNumber r3 { 2, 1 };
19    REQUIRE(r3.numerator() == 2);
20    REQUIRE(r3.denominator() == 1);
21    // After construction a rational number is in its canonical form.
22    RationalNumber r4 { -2 * 3 * 5, -2 * 3 * 7 };
23    REQUIRE(r4.numerator() == 5);
24    REQUIRE(r4.denominator() == 7);
25 }
26
27 TEST_CASE("arithmetic methods", "[rational_number]")
28 {
29     using namespace rational_number;
30     RationalNumber a { 4, 7 },
31                  b { 2, 7 };
32     SECTION("add")
33     {
34         a.add(b);
35         REQUIRE(a.numerator() == 6);
36         REQUIRE(a.denominator() == 7);
37     }
38     SECTION("subtract")
39     {
40         a.subtract(b);
41         REQUIRE(a.numerator() == 2);
42         REQUIRE(a.denominator() == 7);
43     }
44     SECTION("multiply")
45     {
46         a.multiply(b);
47         REQUIRE(a.numerator() == 8);
48         REQUIRE(a.denominator() == 49);
49     }
50     SECTION("divide")
51     {
52         a.divide(b);
```

```

53     REQUIRE(a.numerator() == 2);
54     REQUIRE(a.denominator() == 1);
55 }
56 }
57
58 TEST_CASE("output", "[rational_number]")
59 {
60     using namespace std;
61     using namespace rational_number;
62     // String stream for serializing correct rational numbers
63     ostringstream oss { };
64     SECTION("serialize 1")
65     {
66         RationalNumber { }.output(oss);
67         REQUIRE(oss.str() == "(0/1)");
68     }
69     SECTION("serialize 2")
70     {
71         RationalNumber { 2 }.output(oss);
72         REQUIRE(oss.str() == "(2/1)");
73     }
74     SECTION("serialize 3")
75     {
76         RationalNumber { 2 * 3 * 5, 3 * 5 * 7 }.output(oss);
77         REQUIRE(oss.str() == "(2/7)");
78     }
79     SECTION("serialize 4")
80     {
81         RationalNumber { 2 * 3 * 5, -3 * 5 * 7 }.output(oss);
82         REQUIRE(oss.str() == "(-2/7)");
83     }
84     SECTION("serialize 5")
85     {
86         RationalNumber { -2 * 3 * 5, 3 * 5 * 7 }.output(oss);
87         REQUIRE(oss.str() == "(-2/7)");
88     }
89     SECTION("serialize 6")
90     {
91         RationalNumber { -2 * 3 * 5, -3 * 5 * 7 }.output(oss);
92         REQUIRE(oss.str() == "(2/7)");
93     }
94     SECTION("serialize 7")
95     {
96         RationalNumber { 2, 5 }.output(oss);
97         RationalNumber { 3, -7 }.output(oss);
98         RationalNumber { -11, 13 }.output(oss);
99         RationalNumber { -17, -19 }.output(oss);
100        REQUIRE(oss.str() == "(2/5)(-3/7)(-11/13)(17/19)");
101    }
102 }
103
104 TEST_CASE("input", "[rational_number]")
105 {
106     using namespace std;
107     using namespace rational_number;
108     // String stream with exact and slightly varied serialized rational numbers.
109     // Partially, a robustness test.
110     // RationalNumber::input(istream&) does nit call normalize(!)
111     istringstream iss
112         { "(1/2)(-1/2) (1/-2) (-1/-2) ( 2/ 4) ( -2 / -4) (0/0) " };
113     RationalNumber r;
114     // (1/2) --> (1/2)
115     r.input(iss);
116     REQUIRE(r.numerator() == 1);
117     REQUIRE(r.denominator() == 2);
118     // (-1/2) --> (-1/2)
119     r.input(iss);

```



```

120 REQUIRE(r.numerator() == -1);
121 REQUIRE(r.denominator() == 2);
122 // (1/-2) --> (1/-2)
123 r.input(iss);
124 REQUIRE(r.numerator() == 1);
125 REQUIRE(r.denominator() == -2);
126 // (-1/-2) --> (-1/-2)
127 r.input(iss);
128 REQUIRE(r.numerator() == -1);
129 REQUIRE(r.denominator() == -2);
130 // ( 2/ 4) --> (2/4)
131 r.input(iss);
132 REQUIRE(r.numerator() == 2);
133 REQUIRE(r.denominator() == 4);
134 // ( -2 / -4) --> (-2/-4)
135 r.input(iss);
136 REQUIRE(r.numerator() == -2);
137 REQUIRE(r.denominator() == -4);
138 // (0/0) --> (0/0)
139 r.input(iss);
140 REQUIRE(r.numerator() == 0);
141 REQUIRE(r.denominator() == 0);
142 }
143
144 TEST_CASE("toLongDouble", "[rational_number]")
145 {
146     using namespace rational_number;
147     RationalNumber r { -22, -7 };
148     REQUIRE(r.toLongDouble() == Approx(3.14285));
149 }

```

Listing 122: `RatNumRefSemOp/src/rational_number_test.cpp`

A first difference from Listing 121 is the inclusion of the `<sstream>` header file. This is necessary because this header file contains the definition of the input and output string streams `istringstream` and `ostringstream` introduced earlier.

The first test case tests only the constructor of `RationalNumber`. Therefore it is called "constructor". The tag "[rational_number]" is used for all test cases in this file. The strategy for testing the constructor is different from that for `sign()`. First, it tests whether the expected default values for its parameters are used correctly. Since the member variables `RationalNumber::m_numerator` and `RationalNumber::m_denominator` are private, the corresponding getter methods are used to retrieve their values. There are three ways to call the constructor:

1. As a standard constructor without argument,
2. as a type conversion constructor with only one argument, and
3. with explicit specification of both arguments.

Finally, there is an indirect test of `RationalNumber::normalize()`. Since the implementation of the constructor is available, it is known that this is the only place where this private member function is called. Two prime number products are used to test whether the execution of this function returns the desired result. This concludes the test, as the constructor does no further calculations beyond calling `RationalNumber::normalize()`.

The test case named "arithmetic_methods" introduces a novelty, namely *sections*. In a section, the resources previously defined in the test case are reused. At the beginning of the test case "arithmetic_methods" the variables a and b are declared as RationalNumbers and initialized accordingly. The first section is declared with the macro SECTION() and its name "add" as a parameter. Within the section, the variables a and b are used to perform the actual test. The next section, tagged "multiply", again uses a and b. It is important to note that a and b are not in the state they assumed in the previous section. Rather, a and b have the state in which they were initialized at the beginning of the test case. This is important so that the tests in the sections can be run independently. Consequently, any section can be deleted or moved to any location in the test case, or a new section can be introduced, without affecting the other sections of the test case.

By setting up the required environment for the execution of each test, the sections help to avoid redundancy in the tests.

The "arithmetic_methods" test case only runs happy path tests. It does not contain test data with extreme values. This makes sense at first, since the arithmetic member functions do not contain code to detect and handle errors, such as arithmetic underflow or overflow.

The test case named "output" again makes use of sections. First, it declares an exemplar of ostream named oss. As explained earlier, an ostream is a cout-like stream that stores its data internally in a human-readable form as a string. Its member function ostream::str() provide access to this data. The sections perform an extreme value test ("serialize 1"), a happy path test ("serialize 2"), tests of negative sign permutations ("serialize 3" to "serialize 6"), and a more complex output test ("serialize 7"). Since RationalNumber::output() is declared as a const member function, it can also be called for temporary objects. The statement RationalNumber { 2 }.output(oss); first creates an unnamed temporary exemplar of RationalNumber, i.e. RationalNumber { 2 } returns the *temporary* object. Second, using the dot operator, the member function output() is called for it, i.e. *temporary*.output(oss);.

The next test case, "input", performs a mixture of happy path and robustness tests. The istream iss is initialized with correctly serialized values of RationalNumber, but also contains mismatches. There are syntactic errors, for example extra whitespaces outside parentheses, extra whitespaces inside parentheses. And there are semantic errors, for example, rational numbers with negative denominator and the value (0/0). All these erroneous data are deserialized as expected. So these tests show that RationalNumber::input() can handle erroneous data. This quality of software is called *robustness*. Of course, some limitations apply. A whitespace between a sign and digits cannot be processed

correctly, and a semantically invalid value such as (0/0) can lead to consequential errors.

The last test case performs a happy path test for `RationalNumber::toLongDouble()`. It initializes `r` with -22 and -7, which yields the rational number 22/7. Calculating the decimal value gives a number with decimal places. Since calculations with floating point numbers have a limited range and precision, testing for equality is inappropriate in most cases. Therefore, the comparison for equality is performed using the `Approx` wrapper class. The equality operator `==` has been overloaded for `Approx` to perform a tolerant comparison. `Approx` provides several customization points, which are described in the *Catch2* documentation.

5.6.5. Testing Operators For RationalNumber

Listing 123 shows the tests for the operators related to `RationalNumber`. They introduce only one new feature, namely *nesting of sections*. Here, sections are nested not only for resource reuse, but also for the purpose of a fine-granular test structure.

```
1 // rational_number_operators_test.cpp by Ulrich Eisenecker, June 26, 2021
2
3 #include <sstream>
4 #include "rational_number.hpp"
5 #include "rational_number_operators.hpp"
6
7 TEST_CASE("operator", "[rational_number_operators]")
8 {
9     using namespace rational_number;
10    RationalNumber a { 1, -5 },
11                b { -2, 3 };
12    SECTION("operator+")
13    {
14        SECTION("RationalNumber + RationalNumber")
15        {
16            RationalNumber r { a + b };
17            REQUIRE(r.numerator() == -13);
18            REQUIRE(r.denominator() == 15);
19        }
20        SECTION("RationalNumber + int")
21        {
22            RationalNumber r { a + 2 };
23            REQUIRE(r.numerator() == 9);
24            REQUIRE(r.denominator() == 5);
25        }
26        SECTION("int + RationalNumber")
27        {
28            RationalNumber r { 3 + b };
29            REQUIRE(r.numerator() == 7);
30            REQUIRE(r.denominator() == 3);
31        }
32    }
33    SECTION("operator-")
34    {
35        SECTION("RationalNumber - RationalNumber")
36        {
37            RationalNumber r { a - b };
38            REQUIRE(r.numerator() == 7);
```

```

39     REQUIRE(r.denominator() == 15);
40 }
41 SECTION("RationalNumber - int")
42 {
43     RationalNumber r { a - 2 };
44     REQUIRE(r.numerator() == -11);
45     REQUIRE(r.denominator() == 5);
46 }
47 SECTION("int - RationalNumber")
48 {
49     RationalNumber r { 3 - b };
50     REQUIRE(r.numerator() == 11);
51     REQUIRE(r.denominator() == 3);
52 }
53 }
54 SECTION("operator*")
55 {
56     SECTION("RationalNumber * RationalNumber")
57     {
58         RationalNumber r { a * b };
59         REQUIRE(r.numerator() == 2);
60         REQUIRE(r.denominator() == 15);
61     }
62     SECTION("RationalNumber * int")
63     {
64         RationalNumber r { a * 2 };
65         REQUIRE(r.numerator() == -2);
66         REQUIRE(r.denominator() == 5);
67     }
68     SECTION("int * RationalNumber")
69     {
70         RationalNumber r { 3 * b };
71         REQUIRE(r.numerator() == -2);
72         REQUIRE(r.denominator() == 1);
73     }
74 }
75 SECTION("operator/")
76 {
77     SECTION("RationalNumber / RationalNumber")
78     {
79         RationalNumber r { a / b };
80         REQUIRE(r.numerator() == 3);
81         REQUIRE(r.denominator() == 10);
82     }
83     SECTION("RationalNumber / int")
84     {
85         RationalNumber r { a / 2 };
86         REQUIRE(r.numerator() == -1);
87         REQUIRE(r.denominator() == 10);
88     }
89     SECTION("int / RationalNumber")
90     {
91         RationalNumber r { 3 / b };
92         REQUIRE(r.numerator() == -9);
93         REQUIRE(r.denominator() == 2);
94     }
95 }
96 }
97
98 TEST_CASE("operator<<", "[rational_number_operators]")
99 {
100     using namespace rational_number;
101     std::ostringstream oss;
102     RationalNumber r { 24, 7 };
103     oss << r;
104     REQUIRE(oss.str() == "(24/7)");
105 }

```

```

106
107 TEST_CASE("operator>>", "[rational_number_operators]")
108 {
109     using namespace rational_number;
110     std::istringstream iss { "(24/7)" };
111     RationalNumber r { };
112     iss >> r;
113     REQUIRE(r.numerator() == 24);
114     REQUIRE(r.denominator() == 7);
115 }

```

Listing 123: *RatNumRefSemOp/src/rational_number_operators_test.cpp*

Since arithmetic member functions are tested separately and it is known from the source code that arithmetic operators simply call the corresponding member functions, the correctness of the arithmetic operators is not tested again. This strategy is not optimal because the tests now depend on structural knowledge of the source code. If autonomous implementations for arithmetic operators were later provided, the correctness of their calculations would not be tested at all. Of course, in so-called *glass* or *white-box tests*, knowledge of a unit's implementation can help to implement more comprehensive tests in terms of statement coverage or execution paths. At the moment, the tests focus on one syntactic aspect, namely appropriate type conversions. There is one test case that covers all arithmetic operators. The top-level sections test each operator, and nested sections test calling operators with different combinations of parameter types.

The first nested section tests whether the operator evaluates correctly for two rational numbers, the second tests whether the call with one rational number and one integral value returns the expected result, and the third test does so for one integral value and one rational number.

Finally, there are two test cases for the output operator (stream insertion) and for the input operator (stream extraction). Both tests are happy-path tests. Again, string streams are used for output and input.

5.6.6. Full Test

For testing, the source files *full_test.cpp*, *math_helpers.cpp*, *rational_number.cpp*, and *rational_number_operators.cpp* are compiled and linked to an executable file named *check*. Of course, the executable test file can be named differently. Then all tests are run by executing *./check* in a terminal window. If all tests pass successfully, there is minimal output as Figure 40 shows.

```

=====
All tests passed (70 assertions in 9 test cases)

```

Figure 40: *Report for successful test*

But what happens if a test fails? This can be easily simulated. In *rational_numbers_operators_test.cpp*, test case named "operator +", section named

"int + RationalNumber" the statement `REQUIRE(r.numerator() == 7);` is changed to `REQUIRE(r.numerator() == 6);` . Rebuilding and executing *check* now produces the output shown in Figure 41.

```
-----
check is a Catch v2.13.6 host application.
Run with -? for options

-----
operator
  operator+
  int + RationalNumber
-----
src/rational_number_operators_test.cpp:26
.....

src/rational_number_operators_test.cpp:29: FAILED:
  REQUIRE( r.numerator() == 6 )
with expansion:
  7 == 6

=====
=====
test cases: 9 | 8 passed | 1 failed
assertions: 69 | 68 passed | 1 failed
```

Figure 41: Report for failed test

Catch2 now provides detailed information about the failed test. This way it is easy to locate the failed test see what is the expected value and what is the actual value.

It should be emphasized that a failed test can be caused either by a bug in the software under test or by a bug in the test itself. Therefore, both sites must be thoroughly checked for errors.

The executable test file can be called without options, as has been shown before. It can also be called with a variety of command line options. For example, `./check "[math_helper]"` executes only tests tagged as "`math_helper`", `./check "arithmetic methods"` executes only the "`arithmetic methods`" test case. The available command line options are detailed in the *Catch2* documentation.

It should be mentioned that this example gives only a small insight into testing from a technical perspective. Testing is a separate field in software engineering. Much more can and should be done, there are different testing strategies, and there are many approaches to creating tests and test data. For brevity, the tests shown have not been documented. In practice, it is not unusual to document the tests in detail as well. It should be emphasized again that sometimes there is much more test code than code under test.

5.6.7. Makefile And Testing

Since testing is an essential activity, it is common practice to integrate it into the build process. Listing 124 shows a makefile that includes testing.

```
1 # Makefile for RatNumRefSemOp, January 17, 2023, Author: Ulrich Eisenecker
2
3 CXX = g++
4 CXXFLAGS = -std=c++20
5 SRCDIR = src
6 OBJDIR = obj
7
8 # Declare "doc" and "clean" as phony
9 .PHONY: doc clean
10
11 # Rule 1: Define target "all" with dependencies "demo" and "check"
12 all: demo check
13
14 # Rule 2: Define target "demo" with dependency "$(OBJDIR)/demo"
15 demo: $(OBJDIR)/demo
16
17 # Rule 3: Define target "check" with dependency "$(OBJDIR)/check"
18 check: $(OBJDIR)/check
19
20 # Rule 4: Link "demo"
21 $(OBJDIR)/demo: $(OBJDIR)/main.o $(OBJDIR)/rational_number_operators.o \
22 $(OBJDIR)/rational_number.o $(OBJDIR)/math_helper.o
23 $(CXX) $(CXXFLAGS) -o $(OBJDIR)/demo $(OBJDIR)/main.o \
24 $(OBJDIR)/rational_number_operators.o \
25 $(OBJDIR)/rational_number.o $(OBJDIR)/math_helper.o
26
27 # Rule 5: Compile main.cpp
28 $(OBJDIR)/main.o: $(SRCDIR)/main.cpp$(SRCDIR)/rational_number.hpp \
29 $(SRCDIR)/rational_number_operators.hpp
30 $(CXX) $(CXXFLAGS) -o $(OBJDIR)/main.o -c $(SRCDIR)/main.cpp
31
32 # Rule 6: Compile math_helper.cpp
33 $(OBJDIR)/math_helper.o: $(SRCDIR)/math_helper.cpp \
34 $(SRCDIR)/math_helper.hpp
35 $(CXX) $(CXXFLAGS) -o $(OBJDIR)/math_helper.o \
36 -c $(SRCDIR)/math_helper.cpp
37
38 # Rule 7: Compile rational_number.cpp
39 $(OBJDIR)/rational_number.o: $(SRCDIR)/rational_number.cpp \
40 $(SRCDIR)/rational_number.hpp $(SRCDIR)/math_helper.hpp
41 $(CXX) $(CXXFLAGS) -o $(OBJDIR)/rational_number.o \
42 -c $(SRCDIR)/rational_number.cpp
43
44 # Rule 8: Compile rational_number_operators.cpp
45 $(OBJDIR)/rational_number_operators.o: \
46 $(SRCDIR)/rational_number_operators.cpp \
47 $(SRCDIR)/rational_number_operators.hpp $(SRCDIR)/rational_number.hpp
48 $(CXX) $(CXXFLAGS) -o $(OBJDIR)/rational_number_operators.o \
49 -c $(SRCDIR)/rational_number_operators.cpp
50
51 # Rule 9: Link "check" and execute it
52 $(OBJDIR)/check: $(OBJDIR)/full_test.o \
53 $(OBJDIR)/rational_number_operators.o \
54 $(OBJDIR)/rational_number.o $(OBJDIR)/math_helper.o
55 $(CXX) $(CXXFLAGS) -o $(OBJDIR)/check $(OBJDIR)/full_test.o \
56 $(OBJDIR)/rational_number_operators.o \
57 $(OBJDIR)/rational_number.o $(OBJDIR)/math_helper.o
58 $(OBJDIR)/check
59
60 # Rule 10: Compile full_test.cpp
```

```

61 $(OBJDIR)/full_test.o: $(SRCDIR)/full_test.cpp \
62 $(SRCDIR)/rational_number_operators_test.cpp \
63 $(SRCDIR)/rational_number_test.cpp $(SRCDIR)/math_helper_test.cpp
64 $(CXX) $(CXXFLAGS) -o $(OBJDIR)/full_test.o \
65 -c $(SRCDIR)/full_test.cpp
66
67 # Rule 11: Delete all binaries and executable
68 clean:
69 rm -f $(OBJDIR)/*
70 rm -rf doc/*
71
72 # Rule 12: Generate documentation
73 doc:
74 doxygen

```

Listing 124: *RatNumRefSemOp/makefile*

Besides the integration of tests, it uses some features that have not been introduced yet. They are all explained below.

First, `doc` and `clean` are declared as phony targets. This has the consequence that these targets are made even if files with the same name exist. Then a new target `all` is declared, which depends on `demo` and `check`. Each of these targets depends on files with the same name that may exist in the *obj* subdirectory. By specifying these targets and `all` as the first target, there are the following variations of calling *make* for this makefile:

- *make* – makes *demo* and *check* if needed, and runs *check* when freshly built.
- *make all* – makes *demo* and *check* (like *make* without options).
- *make demo* – makes only *demo* if required
- *make check* – makes only *check* if required, and runs *check* if it is freshly built.
- *make doc* – creates the documentation.
- *make clean* – remove all files contained in the *obj* and *doc* subdirectories; the content of *doc* and all of its subdirectories are deleted recursively.

To make the makefile easier to understand, implicit rules have been omitted, which is unusual for makefiles in practice.

As already introduced (Section [Make](#)), long lines have been split into several lines. Each line that ends with `\` is continued in the next line.

All rules have been documented accordingly, so their purpose should be obvious.

In rule 12 *Doxygen* is called without parameters. For this reason the parameterization file was named *Doxyfile*. One detail must be pointed out about the *Doxyfile* (Listing 125). The line that starts with `EXCLUDE` is new. It excludes *src/catch.hpp* and *src/full_test.cpp* from the documentation creation process. If this line is not present or converted to a comment, a lot of documentation is generated for these otherwise excluded files. Regarding the generated documentation, it is

worth mentioning that the operator calls are missing from the call graph of `main()` (*Doxygen 1.9.1*).

```
1 # Doxyfile for RatNumberRefSemOp for Doxygen 1.9.1
2 PROJECT_NAME           = RatNumberRefSemOp
3 OUTPUT_DIRECTORY      = doc
4 INPUT                  = src/main.cpp src
5 EXCLUDE                = src/catch.hpp src/full_test.cpp
6 GENERATE_LATEX         = NO
7 HAVE_DOT               = YES
8 CALL_GRAPH             = YES
9 CALLER_GRAPH           = YES
10 GRAPHICAL_HIERARCHY   = YES
11 DIRECTORY_GRAPH       = YES
```

Listing 125: RatNumRefSemOp/Doxyfile

The subdirectory *RatNumValSemOp* contains the corresponding files for a value-based design of `RationalNumber`. Since their equivalents in the reference-based design have already been explained in detail, they should be understandable without further explanation.

6. Motivation for the Case Study

So far, both programming and accompanying or subsequent activities such as documentation and testing have been presented. As shown in Figure 42, the focus has been on the solution space. Concepts of an imperative programming language and their representation in C++ as well as selected parts of C++ libraries have been presented. The same applies to additional activities of an individual programmer that help to ensure the understandability and maintainability of the implemented code.

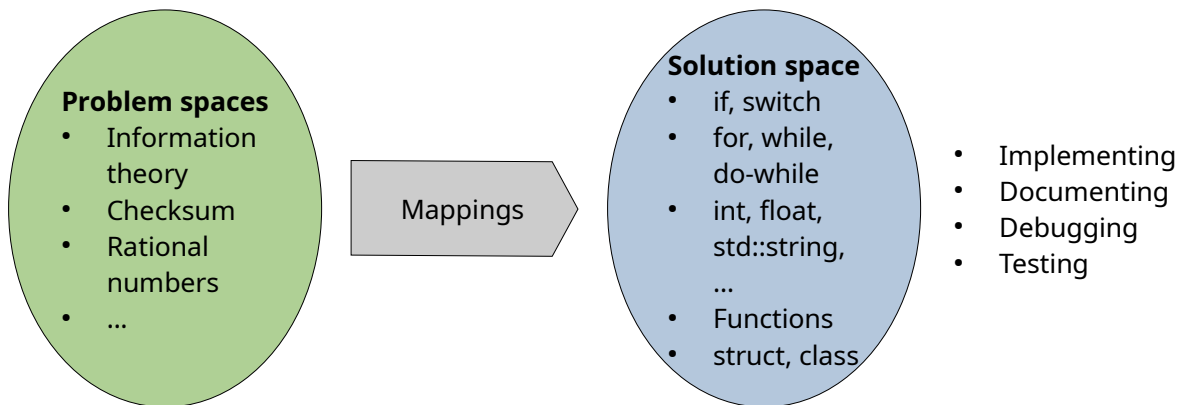


Figure 42: Focus on the solution space

Relatively simple applications were chosen from only a few domains to be implemented as programs. They were introduced in such a way that no effort had to be made to explore and understand them. Consequently, there was also minimal effort for the mappings from problem space to solution space. Either a useful mapping was at hand or it was presented immediately.

Additional emphasis is now placed on the problem space and mapping. Programming is supplemented by essential preparatory activities such as requirements analysis, analysis and design. It is essential to gain knowledge of what is to be developed and to have an idea of how it might be developed before beginning actual programming. It will become clear that this is a difficult process. In practice, it is almost impossible – or at least not economically justifiable – to gather comprehensive and consistent requirements in advance of programming in a single step. It will be shown how to elicit requirements iteratively. For those readers unfamiliar with the domain of the case study, the effort to identify the appropriate domain-specific concepts and their characteristics becomes impressively clear. Each resulting program piece must be thoroughly reviewed to determine whether it meets the appropriate requirements, whether it is usable in practice, and it must be documented for future maintenance and servicing activities. Most programs are so extensive and complex that it is necessary to go through the aforementioned

activities several times in approximately this order. From a practical point of view, the task chosen for the case study is relatively small.

Nevertheless, accomplishing this task will give an intense sense of how challenging this process is. New programming concepts are introduced during the case study. Unlike the assumed reader of this introduction, experienced programmers have an in-depth knowledge of a programming language and its libraries. They also know what resources are available to obtain more information and how to effectively search those resources for the information they need. The assumed reader, of course, lacks this knowledge and experience. For this reason, some coarse-grained heuristics are presented on how to come up with ideas for useful concepts. In addition, the required concepts are once again presented in an authoritative manner. Therefore, the emphasis on mapping is not as strong as it could be. Debugging, testing, and documentation are almost neglected in the case study, which of course is not representative of programming practice.

6.1. A Rule-based Inference Engine

Currently, *artificial intelligence* (AI for short) is experiencing a boom. Today, AI is primarily about *machine learning* (ML for short). A few decades ago, there was also a boom in AI. While ML was still in its infancy, the focus was on capturing human knowledge piecemeal and in a form closer to natural language. This involved finding and using a form of knowledge representation that could be easily created and understood by humans and yet automatically processed by computers, namely *expert systems*.

An *expert system* attempts to mimic the reasoning of a human expert in a particular domain, such as diagnosing a disease, determining the species of an animal, or assessing risk on the road. A *rule-based expert system* uses *rules* to represent knowledge and an *inference engine* to process those rules. An *expert system shell* consists of an inference engine and the ability to load and process an arbitrary, adequately formatted *knowledge base*. A knowledge base bundles rules and *questions* related to a specific domain.

The subject of this case study is the analysis and re-implementation of such a rule-based inference engine.

ESIE™ version 1.1 is a simple rule-based inference engine released as *shareware* by *Lightwave Consultants* in 1985. ESIE™ is an acronym for *Expert System Inference Engine*.

At the time of writing, there are two sources for ESIE™, namely (*The Programmer's Corner* «ESIE.ZIP» *Miscellaneous Language Source Code*, n.d.) and (*Package: Areas/Expert/Systems/Esie/*, n.d.). The first source seems to require registration

before downloading, the second source offers different archives, of which the *.zip* archive seems to be broken, but the *.tar* archive works fine.

The archives from both sources differ in terms of the files they contain. Both contain an *ESIE.COM* executable file. Although both executables differ in when they were created, they identify themselves as version 1.1 in the intro screen after startup. The first source contains the text files *MANUAL* and *READ.ME*. The second source contains the corresponding but different files *manual.esi* and *readme.esi*. The latter files are newer and have a different size. The text file *TUTOR* is included only in the first source and identifies the author as *Edward Reasor*, who owns and operates *Lightwave Consultants*. Although *TUTOR* is a relevant document, it seems safer to use the second source to download *ESIE™* if needed. It must be emphasized that downloading information from the Internet is always at one's own risk and can be potentially harmful.

ESIE™ is mentioned in several publications, for example in (Hecker, 2009), (Yohe, 1987) and (Sciences, 1993). Apparently, it had been used for exploratory and didactic purposes.

To run *ESIE™* on a modern computer, an *emulator* for the MS-DOS operating system is required, e.g. *DOSBox* (*DOSBox, an X86 Emulator with DOS*, n.d.). The installation and configuration of *DOSBox* differs for *Microsoft Windows*, *Linux*, and *macOS* and would go beyond the scope of this text. Nevertheless, one important aspect should be mentioned. To use a German keyboard layout, the line "*keyb de453*" should be added to the initialization file of *DOSBox*.

The goal of this case study is to reimplement a rule-based expert system shell so that it processes the same knowledge bases as *ESIE™*, produces identical results, and exhibits the same observable behavior. The *ESIE™* source code is not available for this task. The only sources of information are *ESIE™* itself, the knowledge bases contained in the archives, newly created knowledge bases, and the companion documents.

The new expert system shell is to be called *EC*, which can be interpreted as *ESIE™ cover version*. However, it is a completely independent software system.

Re-engineering and re-implementing an old software system, also called *legacy system*, is a typical software development task. Often the documentation and expertise on the legacy system is no longer available, and the source code may also be missing, but the legacy system is still relevant to its users and sometimes even critical to business. Fortunately, *ESIE™* is not mission critical, but all other aspects apply to the task of its re-implementation.

This case study also focuses on requirements analysis, analysis, design, and their relationship to programming. Therefore, implementation will be completed once a program is available that behaves exactly – or at least almost exactly – as *ESIE™*.

This program will not meet the criteria for good design. It will be neither documented nor subjected to thorough unit testing. Its main purpose is to develop an idea of how *ESIE*[™] might work and to translate that idea into an executable. One of many ways for re-engineering an existing application is presented in detail. In addition, some new programming concepts are introduced, and last but not least, basic knowledge about rule-based expert systems will be presented.

In fact, the re-implementation of *ESIE*[™] was performed in a very iterative way. The various software engineering activities were performed multiple times, for the data to be represented, for loading a knowledge base, for testing the correctness of the loading and the appropriateness of the chosen data structures (not shown in detail below), for user interaction, and for processing a knowledge base. To present all of this in actual order and iteration would require a lot of text and could potentially be confusing. For didactic reasons, the order of activities has been carefully streamlined and the number of iterations reduced – all from the perspective of an omniscient author. Nevertheless, the impression can sometimes arise that there is an arbitrary switching between activities and topics. This gives a taste of what can happen in the practice of software development. It would create a false image to present everything in a final and polished state.

6.2. Requirements Analysis

There are several sources for requirements elicitation, e.g. domain experts, domain specific documents, legacy systems, and legacy system documents. The re-engineering and re-implementation of *ESIE*[™] in this text will focus on the latter two sources. First, documents related to *ESIE*[™] are presented and evaluated. Second, a sample knowledge base is created that can be processed by *ESIE*[™]. Later, other knowledge bases will be created to further explore the requirements related to *ESIE*[™]. This text does not accurately reflect how to become familiar with the requirements. In practice, it is not mandatory to read the system documents completely first and then try out the system. Usually, one will read some text first and then try out the system, or the other way around. This is repeated until a sufficient understanding of how to use the system is achieved. However, the goal is not to become an expert in the use of *ESIE*[™], but to understand this legacy system to the point where it can be re-implemented. Therefore, great care must be taken in the requirements analysis.

In the following text, domain-specific terms relevant to the analysis are written in *highlighted font*. Terms related to design or implementation are set in code font, as is the case with source programs.

6.2.1. System Documentation

In the following, *manual.esi* (*MAN* for short), contained in (*Package: Areas/Expert/Systems/Esie/*, n.d.) is used for information. *manual.esi* has about the same size as *MANUAL*, contained in (*The Programmer's Corner «ESIE.ZIP» Miscellaneous Language Source Code*, n.d.). *TUTOR* (*TUT* for short), also included in (*The Programmer's Corner «ESIE.ZIP» Miscellaneous Language Source Code*, n.d.), is more extensive and adds some details not included in *manual.esi*. Nevertheless, there is considerable overlap between *MAN* and *TUT*. How to design and write down a knowledge base is a focus of *TUT*.

The first activity for collecting possible requirements is to read *MAN* and copy text phrases that seem to be candidates for requirements. Ideally, such a phrase only addresses one simple requirement. Then, the phrase should be characterized by a few keywords. Simple tools for creating such a requirements document are a word processor or spreadsheet. Of course, there are also professional tools. However, they are not presented here.

In Table 21, these phrases and keywords are recorded in two columns. The left column contains quotations from the documents that are not explicitly formatted as such for simplicity. The citations contain additional text formatted in *italics with blue background*. Square brackets [], enclose additional text for clarification, which may include the ellipsis, ..., to indicate omitted text. To find the exact position of a text phrase, one can search for in the original document.

Quotations from MAN	Keywords
Any word processor will do, but the word processor needs to be capable of producing flat or ASCII, files.	Knowledge Base – Format
In whatever editor you choose, the margins and spacing can be set to whatever you choose. ESIE reads from the file in free format. That means that your file can look pretty strange if you want it to. As long as the syntax is in the correct order the file will load properly. If it does not then ESIE will report an error in the knowledge base and return you to DOS.	Knowledge Base – Format
When you have at least the above two files where you want them, simply type in ESIE at the DOS prompt. The ESIE introductory screen will appear.	Use – Start
At the top of the introductory screen is a prompt asking you to supply the file name where the knowledge base may be found. You may enter any name you please. If the file exists, then ESIE will attempt to load that file.	Use – Start
If no loading errors are found, then ESIE will take you to the top level.	Use – Start
If there are errors in loading the KB, then ESIE will list where it found the errors and return you to DOS.	Use – Start
If the file does not exist, then you have the option of trying again. Just hitting the 'Y' or the 'N' key will answer this prompt.	Use – Start
You will know when you have reached the top level by the distinctive ESIE prompt. It looks like this: "==".	Use – Consult
At the top level you have four different command options: TRACE ON, TRACE OFF, GO, and EXIT. While you can have as many leading and trailing blanks as you wish, ESIE is not quite free form on the command line; you must have one and only one space	Use – Consult

Quotations from MAN	Keywords
between the TRACE and YES/NO options. Other than that, ESIE is free form.	
Importantly, ESIE is case insensitive. Caps look just the same as smalls to ESIE. This is also true in the KB and in end user responses. ESIE is case insensitive everywhere.	Format – Use
When you first enter the top level, trace is off by default.	Use – Consult
You can turn trace on using the TRACE ON command.	Use – Consult
Turning trace on will tell the system to constantly keep you informed of what it is currently looking for and what information it has learned.	Use – Consult
You can turn the trace back off again by entering the TRACE OFF command.	Use – Consult
The GO command is the command to tell ESIE to begin a consultation with the KB that was loaded.	Use – Consult
ESIE will continue with this consultation until it is complete or until an error is found in the logic of the KB.	Use – Consult
You may not turn trace on or off once a consultation has begun. If the user types in TRACE ON, TRACE OFF, GO or EXIT in response to a question, then ESIE will treat that as the response to the question.	Use – Consult
Once ESIE has completed the consultation, or found an error in the logic of the KB, it will return you to the top level for additional commands.	Use – Consult
Use the EXIT command in order to leave ESIE and return to DOS.	Use – Terminate
After a consultation is complete, you have the option of entering ANY of the four commands, including GO, again.	Use – Consult
There are five types of rules in ESIE.	Rules – Types
Below is the syntax for the five types of rules in ESIE: [legalanswers is/are <variable> [<variable>]... *] goal is <variable> [if <variable> is/are <value> [and <variable> is/are <value>]... then <variable> is/are <value>]... [question <variable> is/are "<text>"]... answer is/are "<text>" <variable> [...] In the syntax above the '/' means "either or" you must choose between 'is' and 'are' for that one spot. [...] The symbol "..." in the rule definition above means "repeat as often as desired", and that is exactly what you may do, as long as you are not exceeding some maximum. The brackets mean that the item is optional.	Rules – Syntax – Format
BLANKS are the only recognized delimiter in ESIE. There must be at least one blank between any two tokens in the KB. End-of-line and end-of-file are treated like blanks.	Format – Syntax
In ESIE a <variable> or <value> can be ANY string of non-blank characters, including special and extended ASCII characters, up to 40 characters long. Blanks may not appear in a <variable> or <value>.	Format – Rules – Syntax
ESIE will treat small and capital letters identically.	Format – Syntax
You will note that LEGALANSWERS, IF, and QUESTION type rules may not even be included in the KB, although a KB without rules is not going to be of much use.	Rules – Syntax – Format
In a fully-structured KB, in fact, the order of the rules is totally inconsequential, any rule or any rule type may come before or after any other rule or rule type.	Knowledge Base – Syntax
In some KBs the structure is not complete, so it may be wise to place some rules before others.	Knowledge Base – Syntax
The <text> parts of the above rule types may be any of the 256 characters of the extended ASCII character set except for the quotation mark. The quotation mark terminates <text>. Spaces are allowed in <text>, but double quotation marks – to indicate a single – are not.	Format – Rules – Syntax
<text> may be up to 80 characters long	Format – Rules – Syntax

Quotations from MAN	Keywords
<text> [...] may include end-of-line characters, page feeds, or whatever you wish to improve the appearance of your output.	Format – Rules – Syntax
The LEGALANSWERS rule is used to constrict what the end user of your KB can use as responses.	Format – Rules – Syntax
If LEGALANSWERS is omitted, then anything the end user types in is a valid response.	Format – Rules – Syntax
The splat, (*), is used to terminate the LEGALANSWERS rule. Although the splat, by itself, is a totally valid <variable> or <value> we did not think many of you would want that as a valid response, so we use it as a terminator.	Format – Rules – Syntax
The GOAL rule specifies what you are looking for, that is, it determines what this consultation and KB are all about.	Format – Rules – Syntax
The IF rules and the QUESTION rules are used by ESIE to try and find out what the <variable> of the GOAL should be set to.	Rules – Processing
[IF rules]: the <variable> right after the IF is evaluated, and the evaluated value is compared to the <value> if the two are identical, then any AND parts are compared the same way.	Rules – Processing
[IF rules]: Comparison stops on any rule where an IF part or an AND part do not compare identically.	Rules – Processing
[IF rules]: if they [the comparisons] are all identical then the rule succeeds and the <variable> immediately after the THEN is set to the last <value> specified in the rule.	Rules – Processing
The QUESTION rules are evoked by ESIE when it has gone through the entire KB and can find no IF rule that satisfies what it is currently looking for. It then checks to see if it has a question that it can ask the end user so that it may determine what a <variable> should be.	Rules – Processing
The <text> part of the question is displayed and whatever the end user types in is what <variable> is set to, if the response is legal.	Rules – Processing
If LEGALANSWERS has been specified, then the response is checked to see if it is one of legal ones.	Rules – Processing
If it is not a legal response, then ESIE will list the possible responses, and ask the question again. This will continue until the end user has selected one of the legal answers.	Rules – Processing
If it is legal, then the <variable> specified in the QUESTION rule will be set to the response.	Rules – Processing
If LEGALANSWERS has not been specified, then whatever the user types in is legal.	Rules – Processing
The ANSWER rule is used only when ESIE is done. After ESIE has found the <variable> in the goal statement he will display the <text> in the answer statement followed by the <value> evaluated from the <variable> in the ANSWER rule. You will note that you do not have to display the same <variable> as in the GOAL statement, but that you will often want to.	Rules – Processing
Up to 50 <variables> for LEGALANSWERS.	Knowledge Base – Limits
1 and only 1 GOAL.	Knowledge Base – Limits
Up to 400 IF rule lines.	Knowledge Base – Limits
Up to 100 QUESTION rules.	Knowledge Base – Limits
1 and only 1 ANSWER.	Knowledge Base – Limits
An IF rule line is a <variable> <value> pair. For example: if age is under:10 then type.person is child has two rule lines, while: if age is over:18 and status is alive then type.person is living.adult	Knowledge Base – Syntax – Limits

Quotations from MAN	Keywords
has three.	
One thing that I can tell you about rule positioning. When you have a KB that is not fully structured, rule positioning can be important. You need only worry about rule positioning if the <variables> in the conclusions are identical, and the comparators are similar. [...] In general, you only need to worry about rule placement in the IF category, when the <variables> in the conclusion are identical, and the comparators are similar.	Knowledge Base – Clarification
After ESIE has loaded the KB into internal structures, and the GO command has been issued, ESIE then pushes the GOAL onto a stack.	Rules – Processing
The stack contains items to be searched for, currently looking for the one at the top of the stack.	Processing – Internals
Then ESIE looks through the IF rules for a conclusion that matches the current stack.	Processing – Internals
When one is found then ESIE looks through the comparators, one at a time, until one is found to be not equal to its value, or they are all found to be equal.	Rules – Processing
ESIE checks each comparator by pushing it on the stack and continuing in this fashion.	Processing – Internals
When ESIE can not find an IF rule in the KB with a conclusion identical to what is on the stack then ESIE turns to the QUESTIONS and LEGALANSWERS to get information from the end user.	Rules – Processing
Often ESIE does not need to push anything on the stack as the comparator already has a value and it is equal to the value specified in the KB. In this case the rule succeeds and the conclusion <variable> is set to the <value>.	Processing – Internals
In this way ESIE continues to get information and pop search items off the stack until it learns what the GOAL variable is, or until ESIE has searched the entire KB and found nothing that determines what the GOAL <variable> is.	Processing – Internals
[If ESIE found nothing, it] reports an error in the knowledge base and returns you to top level.	Processing – Internals
[If ESIE has determined the GOAL variable, it] ESIE reports the ANSWER <text> and <variable> and then returns you to top level.	Processing – Rules
Quotation From TUT	
A fully structured KB indicates that there is a terminating leaf on every path of the decision tree. In a fully structured KB it is impossible for the rules in the KB to be mixed up or out of order - there is one and only one path to every single goal in the KB.	Knowledge Base – Clarification

Table 21: Requirement candidates extracted from MAN and TUT

Table 21 has more than 70 rows. Their order reflects the order of occurrence of the quotations in MAN. The only exception is the last row, which is an excerpt from TUT. The rows may contain related phrases or duplicates. This suggests creating a new table that does not contain duplicates and groups the phrases semantically. When creating this new table, the focus is on converting the phrases into requirements. In some cases, this means consistently changing the vocabulary, because MAN uses terms for which there are more appropriate terms today. In addition, a requirement should address only one topic, it should be relevant to implementation, and when implemented, it should be easy to test it. As phrases are condensed into requirements, the new table must provide an indication for a requirement's position in a document. Therefore, both the document and the pages where the information can be found are indicated in the rightmost column.

Keywords are analyzed and reformulated into topics that are grouped in a semantically meaningful order.

A look at the keywords shows some overlaps. For example, *Knowledge Base – Format*, *Knowledge Base – Syntax*, *Rules – Syntax – IF*, etc. are all about *Syntax*. Therefore, *Syntax* seems to be a suitable top-level topic. Keywords such as *Use – Start*, *Use – Consult*, and *Use – Terminate* suggest *User Interface* as another top-level topic. Finally, *Processing – Internals* occurs frequently. Therefore, *Processing* might be also a well-suited top-level topic.

Before identifying subtopics, a meaningful ordering of the top-level topics should be developed. Without having implemented the internal representation of a knowledge base and being able to load it into memory, it does not make sense to implement processing of its content or user interactions via the user interface. Therefore, everything related to *Syntax* is first described in the requirements table. After loading a knowledge base, its processing requires user interaction. Therefore, everything related to *User Interface* is in the second group of requirements. The requirements related to *Processing* are the last group in the requirements table.

Now for the subtopics. What could be considered a valid strategy to find and organize subtopics of *Syntax*? For this purpose, it may be useful to choose an order from the general to the specific, or from the whole to its parts. This results in the subtopics *General*, *Knowledge Base*, *GOAL*, *ANSWER*, *LEGALANSWERS*, *IF*, and *QUESTION* in exactly this order.

Subtopics relevant to *User Interface* are *Start* and *Top Level*. Text phrases described with the keywords *Use – Consulting* have been integrated into *Top Level* or into a subtopic of *Processing*. The keyword *Terminate* has been deleted because no further user interaction is required after entering *EXIT* in the *Top Level*.

The subtopics *Start*, *Load Knowledge Base*, *Top Level*, and *Consult* were chosen for *Processing*. The resulting categories should to some extent reflect the qualities of modularization, i.e. the items in a (sub)category should be more cohesive, while categories should be less coupled with each other. In retrospect, a more thorough analysis might could have lead to a more appropriate category system and even to a better formulation of requirements. In practice, however, resources are always limited. Thus, if too much effort is put into requirement analysis, there will not be enough resources left for implementation and testing. Therefore, the available resources must be reasonably allocated to the various necessary activities. To do this successfully, requires a lot of thought and experience. Therefore, for the time being, no further effort is put into preparing the requirements. Later in the development process, it will be necessary to deal with the requirement analysis again.

The left column of Table 22 contains an *ID* as a unique reference for the specific requirement. The ID consists of three numbers separated by dots. The first number indicates the topic, the second number the subtopic and the third number the specific requirement. The third number is not strictly consecutive, but is assigned in increments of 5. This allows requirements found later to be inserted in an appropriate place in the table. The second column indicates the *Topic* and the third column the *Subtopic*. The fourth column contains a concise *Description* of the requirement. This can be either a citation or a summary restatement. The last column contains the *Source* and exact location within the source.

ID	Topic	Subtopic	Description	Source
1.1.0	Syntax	General	The program is not case sensitive.	MAN, p. 13
1.1.5	Syntax	General	<ul style="list-style-type: none"> • <i><variable></i> means the name of a variable • <i><value></i> means the value of a variable • <i><text></i> means a text • ... (the ellipsis) means that the item to its left can be repeated any number of times • <i>[]</i> (a pair of brackets) enclose an optional item • <i>/</i> (slash) separates alternatives 	MAN, pp. 14
1.2.0	Syntax	Knowledge Base	The knowledge base is a flat ASCII file.	MAN, p. 9
1.2.5	Syntax	Knowledge Base	The formatting of the knowledge base (margins, spacing) is irrelevant as long as the syntax is correct.	MAN, p. 10
1.2.10	Syntax	Knowledge Base	In a fully structured knowledge base, each path of the decision tree has a terminating leaf.	TUT, p. 13
1.2.15	Syntax	Knowledge Base	In a fully structured knowledge base, the order of rules is irrelevant.	MAN, pp. 15, p. 18
1.2.20	Syntax	Knowledge Base	If the knowledge base is not fully structured, IF rules with identical conclusions and identical but additional conditions must precede those with fewer conditions.	MAN, pp. 15, p. 18
1.2.25	Syntax	Knowledge Base	Blanks (spaces) are the only valid delimiters. End-of-line and end-of-file are treated as blanks.	MAN, p. 14
1.2.30	Syntax	Knowledge Base	Legal characters for variable names and values are all non-blank characters, including special and extended ASCII characters. <ul style="list-style-type: none"> • Lowercase letters are treated as uppercase letters 	MAN, p. 14f
1.2.35	Syntax	Knowledge Base	The maximum number of characters in a variable name or value is 40.	MAN, p. 14
1.2.40	Syntax	Knowledge Base	A <i><text></i> part is enclosed in quotes "" and can contain any of the 255 characters of the extended ASCII character set except the quote character. There is no way to insert a quotation mark in <i><text></i> .	MAN, p. 15
1.2.45	Syntax	Knowledge Base	A <i><text></i> part may be up to 80 characters long.	MAN, p. 15f
1.2.50	Syntax	Knowledge Base	There are five types of rules: GOAL, ANSWER, LEGALANSWERS, IF, and QUESTION.	MAN, p. 14
1.3.0	Syntax	GOAL	<ul style="list-style-type: none"> • <i>goal is <variable></i> • GOAL must be contained exactly once • The GOAL rule specifies the goal. → Processing • The rules and questions are used to find the <i><value></i> for the <i><variable></i> of the GOAL → Processing 	MAN, pp. 14 – 17
1.4.0	Syntax	ANSWER	<ul style="list-style-type: none"> • <i>answer is/are "<text>" <variable></i> • ANSWER must be contained exactly once 	MAN, pp. 14f, 17
1.5.0	Syntax	LEGALANSWERS	<ul style="list-style-type: none"> • <i>[legalanswers is/are <variable> [variable] ... *] [This must be an error in MAN. LEGALANSWERS specifies legal answers, i.e. <value>s, but not <variable>s.]</i> • LEGALANSWERS can be included at most once • * (splat, star) terminates the LEGALANSWERS rule; thus, it cannot be used as a legal response • LEGALANSWERS restricts possible user input; if set, 	MAN, pp. 15ff

ID	Topic	Subtopic	Description	Source
			checks, whether a user's response to a question is legal → Processing • If LEGALANSWERS is omitted, the user can enter any text as an answer to a question. → Processing	
1.5.5	Syntax	LEGALANSWERS	LEGALANSWERS can have up to 50 <variables>. [See comment in → 1.5.0]	MAN, p. 17
1.6.0	Syntax	IF	<ul style="list-style-type: none"> • <i>[if <variable> is/are <value> [and <variable> is/are <value>]... then <variable> is/are <value>]...</i> • A knowledge base may contain zero IF rules 	MAN, p. 14
1.6.5	Syntax	IF	<ul style="list-style-type: none"> • Each condition <variable> is/are <value> counts as one rule line • The conclusion <i>then <variable> is/are <value></i> counts as one rule line 	MAN, p. 17
1.6.10	Syntax	IF	There can be up to 400 rule lines.	MAN, p. 17
1.7.0	Syntax	QUESTION	<ul style="list-style-type: none"> • <i>[question <variable> is/are "<text>"]...</i> • A knowledge base may contain zero QUESTION rules 	MAN, p. 14
1.7.5	Syntax	QUESTION	A knowledge base can contain up to 100 QUESTION rules.	MAN, p. 17
2.1.0	User Interface	Start	After startup, the program asks for the name of a file that contains a knowledge base. → Processing	MAN, p. 10
2.1.5	User Interface	Start	<p>If a file with the given name does not exist, the program offers to try again with Y and N as possible answers.</p> <ul style="list-style-type: none"> • If the user presses N, the program exits. • If the user presses Y, the program asks again for the name of a file containing a knowledge base. → Processing	MAN, p. 12
2.1.10	User Interface	Start	If a file with the specified name exists, the program tries to load it. → Processing	MAN, p. 12
2.1.15	User Interface	Start	If a syntax error is found in the knowledge base or if an error occurs while loading the knowledge base, the error is reported and the program is terminated. → Processing	MAN, pp. 10, 12
2.1.20	User Interface	Start	After successfully loading a knowledge base, the program enters the top level. → Processing	MAN, p. 12
2.2.0	User Interface	Top Level	==> is the prompt of the top level.	MAN, p. 13
2.2.5	User Interface	Top Level	<p>Valid commands in the top level are:</p> <ul style="list-style-type: none"> • TRACE ON • TRACE OFF • GO • EXIT <p>User input is not case sensitive.</p>	MAN, p. 13
2.2.10	User Interface	Top Level	TRACE ON and TRACE OFF must be entered with exactly one space between TRACE and one of the options ON or OFF.	MAN, p. 13
2.2.15	User Interface	Top Level	When the top level is entered, the trace is switched off by default.	MAN, p. 13
2.2.20	User Interface	Top Level	<p>TRACE ON turns trace on.</p> <ul style="list-style-type: none"> • When trace is on, the program informs what it is looking for and what it has learned → Processing 	MAN, p. 13
2.2.25	User Interface	Top Level	TRACE OFF switches the trace to off.	MAN, p. 13
2.2.30	User Interface	Top Level	EXIT terminates the program.	MAN, p. 13
2.2.35	User Interface	Top Level	<p>GO starts a consultation.</p> <ul style="list-style-type: none"> • After starting a consultation, it is no longer possible to enter top-level commands; only responses to QUESTIONS can be entered → Processing 	MAN, p. 13
2.2.40	User Interface	Top Level	<p>After finishing a consultation or after reporting an error in the knowledge base that occurred during the consultation, the program returns to top level.</p> <p>All top level commands can be used again.</p>	MAN, p. 13

ID	Topic	Subtopic	Description	Source
3.1.0	Processing	Start	The user is asked for the name of a file containing the knowledge base. <ul style="list-style-type: none"> If a file with the specified name does not exist, the program offers to try again or to quit. If a file with the specified name exists, the program tries to load the knowledge base. → User Interface 	MAN, p. 12
3.2.0	Processing	Load Knowledge Base	If an error occurs while loading the knowledge base, the error is reported and the program terminates. → User Interface	MAN, pp. 10, 12
3.2.5	Processing	Load Knowledge Base	If the knowledge base is loaded successfully, the program switches to the top level. → User Interface	MAN, p. 12
3.3.0	Processing	Top Level	Valid commands at the top level are: <ul style="list-style-type: none"> TRACE ON TRACE OFF GO EXIT The input is not case sensitive. Trace is initially switched off. TRACE ON and TRACE OFF switch between trace on and trace off. EXIT terminates the program and GO starts a consultation. → User Interface	MAN, p. 13
3.3.5	Processing	Consult	A consultation continues until it is completed or an error is found in the knowledge base logic.	MAN, p. 13
3.3.10	Processing	Consult	The program manages a stack. <ul style="list-style-type: none"> This stack contains items to be searched <i>[The stack items are actually names of variables that do not have a value yet.]</i> The program is currently searching for the item that is on top of the stack. 	MAN, p. 22
3.3.15	Processing	Consult	At the very beginning of the consultation, the <variable> of GOAL is pushed onto the stack. Then the processing of the knowledge base begins.	MAN, p. 22
3.3.20	Processing	Consult	The GOAL <variable> is the first item on the stack.	MAN, p. 22
3.3.25	Processing	Consult	<ul style="list-style-type: none"> The GOAL rule specifies the goal of the consultation process. The IF rules and QUESTION rules are used to find the <value> on which to set the <variable> of the goal. 	MAN, p. 16
3.3.30	Processing	Consult	As soon as the GOAL <variable> is set, the consultation ends, and the program switches to the top level.	MAN, p. 16
3.3.35	Processing	Consult	Processing continues until the GOAL <variable> has been set or the entire knowledge base has been searched and nothing has been found to determine the <value> of the GOAL <variable>.	MAN, p. 22
3.3.40	Processing	Consult	If the GOAL <variable> was set, the program displays the <text> of the ANSWER rule followed by the ANSWER <variable>. <ul style="list-style-type: none"> GOAL <variable> and ANSWER <variable> need not to be identical, but usually should be. After that the program returns to the top level 	MAN, pp. 17, 22
3.3.45	Processing	Consult	if the <value> of the GOAL <variable> cannot be determined, the program reports an error in the knowledge base and returns to the top level.	MAN, p. 22
3.3.50	Processing	Consult	When processing the knowledge base, the program looks for IF rules whose conclusion matches the current stack. <i>[Obviously, this phrase means the "current top of the stack". Also, the conclusion of an IF rule matches the top of the stack if the variable names of the conclusion and the top of the stack are the same.]</i>	MAN, pp. 16, 22
3.3.55	Processing	Consult	When a IF rule is found with a matching conclusion, its conditions <i>[MAN uses the term "comparison" instead of "condition". Here the term "condition" is preferred because it is more general and more often used.]</i> are checked until one condition is not equal to its value or they are all equal.	MAN, p. 22

ID	Topic	Subtopic	Description	Source
3.3.60	Processing	Consult	Checking a condition means that the <value> of the condition <variable> is compared with the <value> of the condition. If both values are equal, the condition is TRUE, otherwise it is FALSE.	MAN, p. 16
3.3.65	Processing	Consult	<ul style="list-style-type: none"> • If a condition is TRUE, the next condition – if any – is evaluated • If a condition is FALSE, the checking the conditions of a rule is aborted 	MAN, pp. 16, 22
3.3.70	Processing	Consult	If all conditions are TRUE, nothing is pushed to the stack, and the conclusion <variable> is set to the conclusion <value>. The corresponding <variable> is popped from the stack.	MAN, pp. 16, 22
3.3.75	Processing	Consult	If the program does not find no IF rule where the conclusion <variable> bis equal to the <variable> currently searched for, it searches the QUESTION rules for a QUESTION where a <variable> is equal to the <variable> searched for.	MAN, pp. 16, 22
3.3.80	Processing	Consult	If no QUESTION rule is found that sets the <value> of the searched <variable>, the program reports an error in the knowledge base and returns to the top level.	MAN, p. 22
3.3.85	Processing	Consult	<p>If a QUESTION rule is found to set the <value> of the searched <variable>, the QUESTION <text> is displayed to the user and the user is asked to answer the question.</p> <ul style="list-style-type: none"> • Answering the question depends on whether LEGALANSWERS rule is present or not. After answering the question, the QUESTION <variable> is set to the user's input. 	MAN, pp. 16f
3.3.90	Processing	Consult	<p>If the LEGALANSWERS rule is not present, the user can make any input for an answer.</p> <ul style="list-style-type: none"> • Lowercase letters in the input are treated like uppercase letters 	MAN, p. 17
3.3.95	Processing	Consult	<p>If the LEGALANSWERS rule is present, the user's input must be a legal answer, i.e. one of the <variable>s [As pmentioned earlier, LEGALANSWERS specifies <value>s, but not <variable>s. Here, the erroneous term <variable> is used only for consistency.] specified in the LEGALANSWERS rule.</p> <ul style="list-style-type: none"> • If the user's input is not legal, the program displays all <variable>s of LEGALANSWERS and prompts the user again for input. 	MAN, p. 17

Table 22: Consolidated and structured requirements

Clarifications and corrections are enclosed in square brackets [] and formatted in *italics with blue background*. The right arrow, →, indicates a cross-reference within the table.

In professional software development, tools can provide various ways to organize requirements and ways to identify them.

6.2.2. Use of the Legacy System

To promote understanding of ESIE™, a knowledge base is created from scratch, which can then be loaded and consulted in ESIE™.

In the previous section it was said that a fully structured knowledge base corresponds to a binary decision tree where all paths end in a final leaf. In

computer science, there are many knowledge domains that are organized in this way. In the implementation of the cover version of *ESIE™*, the container templates of the *Standard Template Library* (abbr. *STL*) of the *C++* standard library are used. A *container* allows elements of the same type to be stored and managed according to different constraints and requirements. For this reason, the example knowledge base is created to select the most appropriate container from the available *unordered* or *adaptive containers*. (“The *C++* Standard Template Library (*STL*),” 2015) provides a concise overview with two flow charts for container selection. Figure 43 shows a *decision tree* for selecting between unordered or adaptive containers based on the corresponding flowchart shown in (“The *C++* Standard Template Library (*STL*),” 2015).

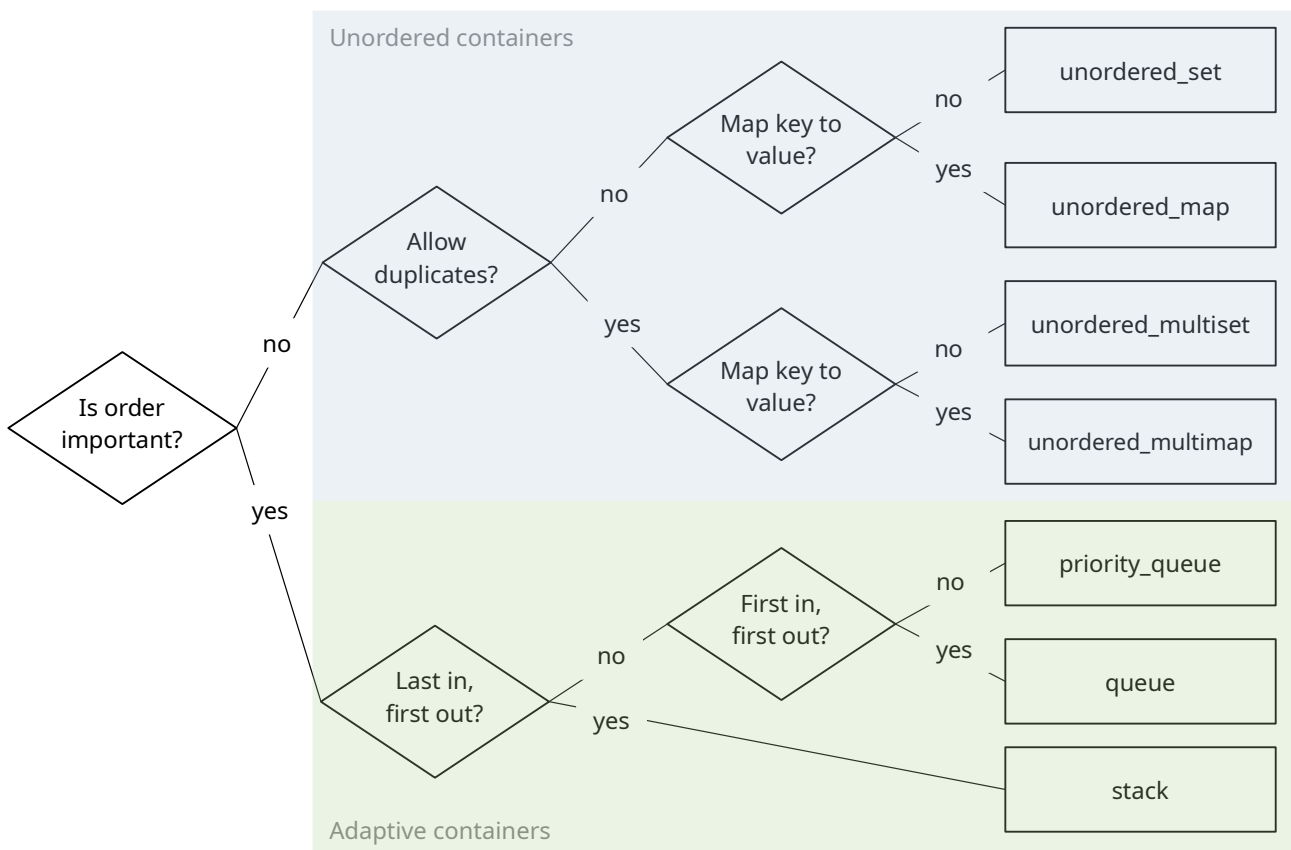


Figure 43: Adaptive and unordered containers of the *C++ STL*

The decision tree diagram consists of *nodes* of different types and *links*. Normally it is read from top to bottom. Here, for layout reasons, the diagram had to be rotated 90 ° counterclockwise. It is therefore to be read from left to right. A diamond-shaped node stands for a decision or a question. Since it is a binary tree, there is one link for the decision *no* and one link for the decision or answer *yes*. A *terminal node*, also called a *leaf*, is indicated by a rectangle containing the proposed *STL* container.

This decision tree can be easily converted into a textual knowledge base for *ESIE™*. The text file containing the knowledge base is named *UN_AD_CN.KB*

(*UNordered_ADaptive_CoNtainers.KnowledgeBase*). The restriction to a file name with a maximum of eight characters is due to the limitations of *MS-DOS*. The extension of the file name is optional and may be a maximum of three characters. Now to the content of the knowledge base.

Obviously, the goal is to determine the container (Listing 126).

```
1 goal is container
```

Listing 126: GOAL rule from UN_AD_CN.KB

Legal answers are limited to yes and no (Listing 127).

```
2 legalanswers are yes no *
```

Listing 127: LEGALANSWERS rule from UN_AD_CN.KB

After determining the container, the *ANSWER* rule is executed (Listing 128). Since the user is only interested in the goal variable, the value of the goal, i.e. container, is output. The associated string clarifies that the automatic recommendation should be used with caution, as the knowledge base may be incorrect or incomplete.

```
3 answer is
4 "The STL container matching your needs is probably a(n) "
5 container
```

Listing 128: ANSWER rule from UN_AD_CN.KB

Then, all decision nodes are coded as questions. Listing 129 shows an example question.

```
6 question order.is.important
7 is "Is order important?"
```

Listing 129: Exemplary QUESTION rule from UN_AD_CN.KB

In total, there are five questions. This is completely consistent with the six diamonds in the decision tree, since one diamond occurs twice.

The rules simply follow all paths and combine all decisions on the path up to the leaf with logical And. In its action part, each rule assigns the inferred value to the variable container. Listing 130 shows an example.

```
8 if order.is.important is no
9 and duplicates.are.allowed is no
10 and map.key.to.value is yes
11 then container is unordered_map
```

Listing 130: Exemplary IF rule from UN_AD_CN.KB

There are seven rules in total, which corresponds to the number of seven possible values for the goal variable.

Listing 131 shows the complete knowledge base.

```
1 goal is container
2
```



```

3 legalanswers are yes no *
4
5 answer is
6 "The STL container matching your needs is probably a(n) "
7 container
8
9 question order.is.important
10 is "Is order important?"
11
12 question duplicates.are.allowed
13 is "Are duplicates allowed?"
14
15 question map.key.to.value
16 is "Do you want to map a key to a value?"
17
18 question last.in.first.out
19 is "Is the element added last the first to be taken out?"
20
21 question first.in.first.out
22 is "Is the element added first the first to be taken out?"
23
24 if order.is.important is no
25 and duplicates.are.allowed is no
26 and map.key.to.value is no
27 then container is unordered.set
28
29 if order.is.important is no
30 and duplicates.are.allowed is no
31 and map.key.to.value is yes
32 then container is unordered.map
33
34 if order.is.important is no
35 and duplicates.are.allowed is yes
36 and map.key.to.value is no
37 then container is unordered.multiset
38
39 if order.is.important is no
40 and duplicates.are.allowed is yes
41 and map.key.to.value is yes
42 then container is unordered.multimap
43
44 if order.is.important is yes
45 and last.in.first.out is no
46 and first.in.first.out is no
47 then container is priority.queue
48
49 if order.is.important is yes
50 and last.in.first.out is no
51 and first.in.first.out is yes
52 then container is queue
53
54 if order.is.important is yes
55 and last.in.first.out is yes
56 then container is stack

```

Listing 131: Complete UN_AD_CN.KB knowledge base

After manually entering the text of the knowledge base into a file, it is highly recommended to enter the *TRACE ON* command in the top level after loading the knowledge base. This helps to identify misspelled variable names, which are a common cause of failed analyses. A complete test of the knowledge base requires as many analyses as there are leaves in the decision tree, of course with the corresponding decisions for the nodes of each path.

6.3. Analysis Model

After an initial iteration of requirement analysis, the creation of an analysis model is the logical follow-up activity. The analysis model maps the requirements to essential concepts and their responsibilities. Usually this is visualized with a *UML* class diagram. The *Unified Modeling Language (UML)* for short) is the de facto standard in object-oriented software development and comprises many diagram types for different purposes. In the following, only the class diagram is used. Required notation elements are introduced as needed.

The purpose of an analysis model is to provide a sufficient functional view of the system, its structure and behavior. Once some of these basics are in place, an appropriate design can be considered. In practice, there will be several iterations between requirement analysis, analysis modeling, design modeling, implementation, and testing. In the following, only the structure is discussed, and small implementation pieces serve as partial design models.

The central concept of the analysis model is the *KnowledgeBase*. It is modeled as a class. A class is like an ADT with some additional properties. It is symbolized by a rectangle with three compartments. The upper compartment contains the *class name*, the middle one contains the *attributes* (they correspond to member data in C++) and the lower one contains the *methods* (they correspond to member functions in C++). A knowledge base consists of rules of five different types. Each rule type is represented as a separate class, namely *LegalAnswers* (corresponding to the *LEGALANSWERS rule*), *Question* (corresponding to the *QUESTION rule*), *Rule* (corresponding to the *IF rule*), *Answer* (corresponding to the *ANSWER rule*) and *Goal* (corresponding to the *GOAL rule*). The chosen class names differ from the names used in MAN. Obviously, all these concepts were called *rules* because they occur in a knowledge base. But as will become clear later, they do not have so much in common that they could all be grouped under the term *rule*.

The knowledge base and each rule class are connected by a so-called *aggregation relationship*, or *aggregation* for short. The significance of an aggregation is that the aggregating object needs the aggregated object to function. When the aggregating object terminates, its aggregated object also terminates, unless the aggregated object is removed first to preserve it. A small unfilled diamond attached to the aggregating object indicates aggregation. On the side of the aggregated object, the so-called *cardinality* is indicated. It tells how many aggregated objects can or must participate in the aggregation. Common values are 0..1, which means that at most one aggregated object is allowed, 1, which means that exactly one aggregated object is required, *, which means that any number of aggregated objects is allowed, and 1..*, which means that at least one aggregated object or any number of them is allowed. In addition, the exact cardinality can be specified with a lower limit and an upper limit, e.g. 0..100, which means that there can be no aggregated object and up to 100

aggregated objects. Aggregation applies to objects, but is represented at the class level.

Figure 44 shows a class diagram with the *KnowledgeBase* class and its aggregated classes.

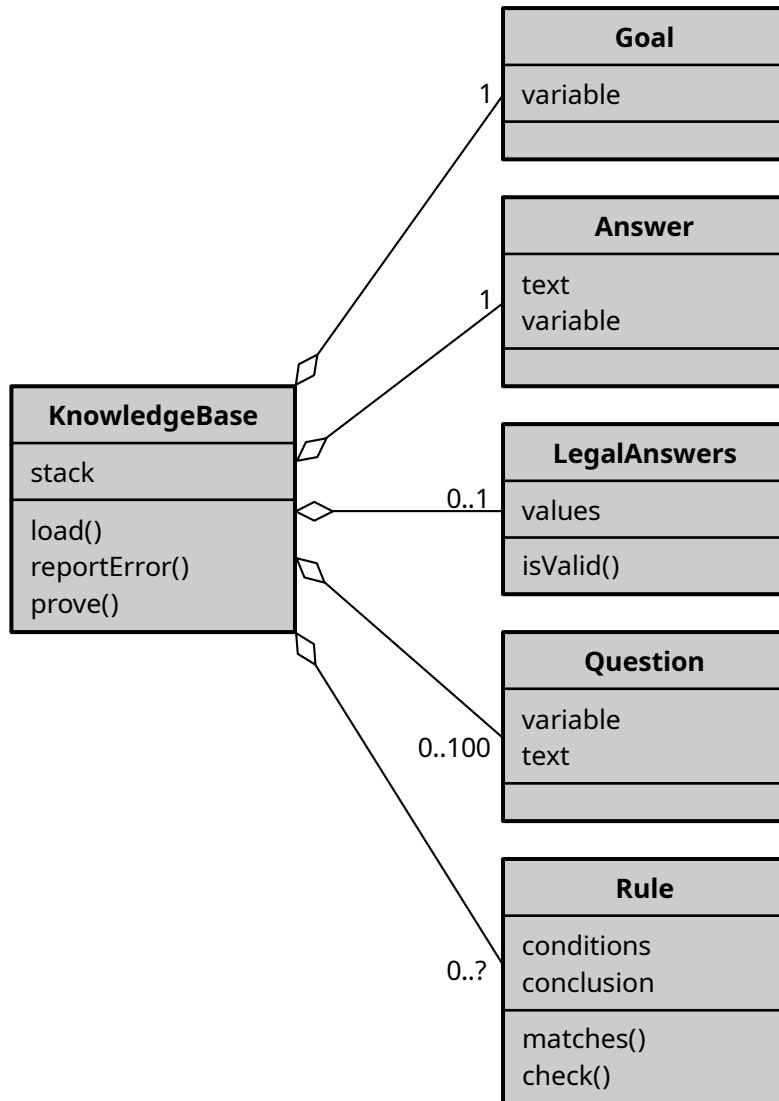


Figure 44: UML class diagram of the *KnowledgeBase* analysis model

In the following, all numbers mentioned in connection with requirements refer to Table 22.

First, a closer look at the *KnowledgeBase* class will be taken. Its only attribute is *stack*. The *stack* is mentioned for the first time in requirement 3.3.10. The requirement states that the *stack* is managed by the program and is essential for processing the content of the knowledge base. Therefore, it has been moved to *KnowledgeBase* for the time being.

Loading is mentioned for the first time in requirement 2.1.15. Obviously, loading a knowledge base is a task best assigned as a method named *load()* to the *KnowledgeBase*. Especially when loading or processing a knowledge base, errors can occur. Therefore, *reportError()* becomes a method of *KnowledgeBase*. Processing knowledge seems to be an obvious task of *KnowledgeBase*. Requirements 3.3.15 and 3.3.20 state that the target is the first element pushed onto the stack. The main purpose of the program is to assign a value to the variable of the goal. Once this is done, the knowledge base consultation is finished. This is why the corresponding method is called *prove()*. It tries to prove that there is a value that can be bound to the goal variable. At the moment there is no evidence for adding more methods or attributes to the *KnowledgeBase* class in the analysis model.

The *Goal* class is mentioned first in requirement 1.3.0. Its only attribute is the name of a *variable*, and it must occur exactly once in a knowledge base. Therefore, it has cardinality 1. *Goal* is a sparse class, but it plays an essential role in knowledge base processing.

The *Answer* class appears for the first time in requirement 1.4.0. It has the attributes *text* and *variable*. A knowledge base must have exactly one answer, denoted by cardinality 1. *Answer* is also a very simple class.

The *LegalAnswers* class is explained in detail in requirement 1.5.0. *KnowledgeBase* may or may not aggregate an exemplar of *LegalAnswers* as indicated by cardinality 0..1. According to MAN, *LegalAnswers* manages up to 50 variables. This statement does not appear to be correct, as the values managed by *LegalAnswers* limit the values entered by the user in response to a question. *LegalAnswers* therefore contains up to 50 *values*. No lower limit is specified. A value is a simple string and as such is not represented as a class in the analysis model. When *LegalAnswers* is present, it checks whether an answer given by the user is valid. This task is assigned to the *isValid()* method.

The *Rule* class represents the *IF Rule* whose syntax is described by the requirements 1.6.0, 1.6.5 and 1.6.10. It consists of *conditions* and a *conclusion*. Since *conditions* and *conclusion* are not modeled as separate classes, further details are not specified until design or implementation. Requirements 3.3.50 and 3.3.55 mention that a rule matches a variable on the stack if its conclusion uses that variable. Therefore, the *matches()* method is present. If a rule matches a variable, then all its conditions are checked, which is described in requirements 3.3.55 and following. The method *check()* serves this purpose. The cardinality of *Rule* is specified as 0...?. Requirement 1.6.0 implicitly states that a knowledge base can contain no rules at all. This explains the lower limit of 0. Requirement 1.6.10 states that a knowledge base may contain a maximum of 400 rule lines. The definition of rule lines is found in requirement 1.6.5, so there is no clear upper bound on the rules aggregated in a

knowledge base. For this reason, the upper limit is indicated by a question mark. This issue must be resolved during design or implementation.

The syntax of the *Question* class is described in requirements 1.7.0 and 1.7.5. A knowledge base can aggregate up to 100 questions. *Question* has the attributes *variable* and *text*. Methods are not currently displayed.

6.4. Design Model

Based on the analysis model of the system structure, the next step is to consider options for implementation in terms of classes, attributes and the relationships between classes. This part is called *Data*. In some cases, it is advisable to focus on the behavior first, i.e. classes and methods. However, a knowledge base is primarily data-driven, and its data is processed in a uniform manner.

After designing the structure, the methods for loading an existing knowledge base are designed. This part is called *Loading*. The last step is the design of processing. This part is called *Processing*.

The aforementioned activities transform the analysis model into a design model, which is then implemented.

6.4.1. Data

KnowledgeBase is the central class of the program. It contains the goal, the answer, the rules, the questions as well as the legal answers together. In the analysis model, *Goal* is modeled as its own class. Its only data member is a variable, which can be thought of as a C++ string. This data member does not change, and *Goal* has no specific behavior or responsibilities. For this reason, it is not designed as a class. Instead, it becomes a data member of KnowledgeBase with the name `m_goal` of type `std::string`.

The *Answer* class of the analysis model has the attributes *text* and *variable*. Both attributes can best be thought of as C++ strings. Nevertheless, they are semantically coupled. This might suggest an implementation as a class. However, the C++ library provides a struct template `std::pair<>` as a viable alternative. ***Templates can best be thought of as forms for classes and functions (template variables are not considered here). They must either be explicitly instantiated with types (class or struct templates) or they deduce the types from the arguments passed to them (function templates).***

To indicate that a name refers to a template, an empty pair of angle brackets is appended to the name. This has no syntactic meaning. It is merely a convention to inform the human reader that the name refers to a template.

`pair<>` is a struct template instantiated with two types, for example `pair<std::string,int>`. The result is a struct with a public data member named `first` of type `std::string` and a public data member named `second` of type `int`. Since *text* and *variable* are both of type `std::string`, the corresponding data member is defined by `KnowledgeBase` as `pair<string,string> m_answer;`. The only drawback is that the names of the parts no longer have any functional meaning. That is, `m_answer.first` means *Answer::text* (analysis model), and `m_answer.second` means *Answer::variable* (analysis model). So care must be taken not to confuse `m_answer.first` and `m_answer.second`. This way two classes of the analysis model disappear in the design.

This is different for the *LegalAnswers* class. It must manage up to 50 values representing legal answers. A single value can again be thought of as a C++ string. Another aspect is that *LegalAnswers* may or may not be present (its cardinality is 0..1). If *LegalAnswers* is not present, the user can type anything in response to a question. Otherwise, a check is made to see if the user's answer is known by *LegalAnswers*. Thus, two sequential checks take place. First, a `KnowledgeBase` would need to check if an exemplar of *LegalAnswers* exists. If so, it must check to see if the user entered a legal answer. Obviously, checking for the presence of a *LegalAnswers* exemplar can be considered a task of the `KnowledgeBase`. But checking whether a string is a legal answer is a primary responsibility of *LegalAnswers*. Splitting a responsibility among multiple classes is not a good idea, as it compromises modularization (cohesion is compromised and coupling may increase). However, managing legal answers directly in the `KnowledgeBase` class would increase its size and complexity. Another alternative is to transfer responsibility entirely to *LegalAnswers*. In this case, *LegalAnswers* will always exist as a data member in `KnowledgeBase`. *LegalAnswers* will then know with certainty whether or not legal answers have been defined in the knowledge base and decide accordingly. For now, it is assumed that a *LegalAnswers* class will be designed for these purposes. Consequently, `LegalAnswers m_legalAnswers;` will be added as a data member to `KnowledgeBase`.

Rule is a more complicated class. It consists of *conditions* and a *conclusion*. Each elementary condition and conclusion corresponds to a *rule line*. A `KnowledgeBase` can contain up to a flexible limit of 400 rule lines. To manage these rules, a container is needed that can store a flexible number of rules. The *C++ Standard Template Library* offers a variety of container templates for different purposes. At the moment, the decision for one of them is postponed. Instead, it is assumed that a *Rules* class, yet to be defined, will serve as a container for rules and their management. This results in a new data member being added to the `KnowledgeBase` class, namely `Rules m_rules;`. This is the first time a new class is added in the design that was not yet present in the analysis model. Basically, this class implements the aggregation relationship from *KnowledgeBase* to *Rule*. The design of

Rule is also deferred. When resuming the design of Rule, the `Rule::matches()` and `Rule::check()` methods are considered.

The *Question* class is similar to the *Rule* class. It is less complex, but there can be no questions or up to 100 questions in a knowledge base. Obviously, a container is also needed to manage questions. This decision about the use of a particular container is again postponed by introducing a yet to be defined *Questions* class, which is responsible for managing questions in a knowledge base. Consequently, another member is added to the *KnowledgeBase* class, namely `Questions m_questions;`. Again, a new class appears in the design that was not present in the analysis model.

So much for the aggregations. The analysis model shows *stack* as an attribute of *KnowledgeBase*. It is needed for processing. Therefore, further discussion of *stack* is deferred to the design and implementation of processing.

Another problem has not been addressed yet. The *Goal*, *Answer*, *Question* and *Rule* classes refer to variables by their names. A *variable* can be thought of as a *pair* consisting of a *name* and a *value*. If the *value* is not set or is empty, the *variable* can be assumed to be defined but not initialized. The *value* of a *variable* can be defined, for example, by a *Question* or by the conclusion of a *Rule*. The *value* of a *variable* can also be queried, for example, by an *Answer* or by the conditions of a *Rule*. Although no class *Variable* or a class for managing variables is part of the analysis model, it is clear that variables must be managed by an entity to be called *Variables*. Otherwise, the knowledge base cannot be processed. For this reason, the member `Variables m_variables;` is added to the *KnowledgeBase* class.

6.4.1.1. Digression

One could argue that the path from requirements to analysis to design is arbitrary. There are two main reasons for this objection.

The first lies in the question of which requirement is mapped to which programming concept. This depends largely on the programming concepts available. In this text, programming concepts have been introduced mostly *on the fly* to solve small programming problems. This text is not a systematic introduction to programming concepts. Therefore, the selection of a particular programming concept for a particular requirement is based on the knowledge and experience of the author of this text. Software engineering is not a subject of the laws of nature, but of the human mind. Therefore, this text serves as a kind of example or model of how to develop programs. At the moment, the chosen concepts for analysis, design and implementation are ADT and object-oriented programming.

The second source is the progressive acquisition of knowledge and understanding while performing a software development task. In addition to the importance of

KnowledgeBase
m_goal : string m_answer : pair<string,string> m_legalAnswers: LegalAnswers m_rules : Rules m_questions : Questions m_variables : Variables

experience, software development is a highly creative activity. In software development, there are many decisions that lead to feedback. All programming concepts, methods, and tools are guidelines that help evaluate decisions and roll back when an error is detected. Visualizing ideas with diagrams, expressing models in code, compiling and running that code, testing, all support this process of knowledge acquisition, making corrections, and perfecting until the desired system is available and meets its specification. This text reflects at least part of this creative process and attempts to provide the means necessary to understand and reproduce it. Today, tools based on generative artificial intelligence can process natural language specifications and respond with code that can be saved as a text file and compiled. However, there is no guarantee that this code will compile without errors and produce the desired result. This code still needs to be reviewed by human programmers, and even if tests and documentation are also generated by artificial intelligence, their validation by humans is essential. To claim that software can be developed more systematically from requirements to implemented system would not be honest.

6.4.1.2. KnowledgeBase

In the first iteration of the design, many decisions were made for classes that impact the KnowledgeBase class. For this reason, Figure 45 shows a *UML* class diagram that summarizes all recent changes to the KnowledgeBase class from the data perspective.

Figure 45: KnowledgeBase class from the data perspective

In *UML*, the name of the attribute comes first. It can be followed by a colon and the type of the attribute. This syntax reverses the order of the *C++* syntax for declaring data members in a class.

6.4.1.3. LegalAnswers

In the next step, the LegalAnswers class is designed. It manages up to 50 values, i.e. *C++* strings. Assuming that legal answers are unique and that a container can store

them in any order, `std::set<>` would be a suitable container template. How can this assumption be checked? Obviously, this is an issue of the design step *Loading* yet to come. In fact, this problem was only solved once the author had implemented loading. After that, the choice of `set<>` had to be revised. To streamline the process a bit and not have to revise the decision later, it is clarified now. For this purpose, another small program for gaining insight is developed and a new feature of operating systems is introduced.

The program shown in Listing 132 creates a syntactically correct knowledge base with 0 rules and 0 questions. There are 51 values for *LEGALANSWERS*, all of which are the same. After compiling the program, its output must be redirected to a text file with a name valid for MS-DOS. Assuming the program was compiled to *gen_kb*, it must be executed as follows:

```
./gen_kb > LA51SAME.KB
```

The chevron, `>`, redirects the standard output, here the output in the console window, to the file with the following name, here *LA51SAME.KB*. This file name is exactly 8 characters long and has an extension of two characters in length. This is within the limits for a file name under *MS-DOS*.

```
1 // 51SameLegalAnswers.cpp by Ulrich Eisenecker, July 27, 2021
2
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     cout << "goal is something" << endl << endl;
9
10    cout << "legalanswers are ";
11    for (auto i { 1 }; i <= 51; ++i)
12    {
13        cout << "same ";
14    }
15    cout << "*" << endl << endl;
16
17    cout << "answer is \"The answer is \" something" << endl << endl;
18
19    // no rules
20    // no questions
21 }
```

Listing 132: *ec/51SameLegalAnswers.cpp*

When loading this knowledge base, *ESIE™* issues the following error messages and terminates:

Too many legalanswers encountered in the LEGALANSWERS rule.

Apparently, *ESIE™* does not check for duplicate values and stores each value.

This suggests that the upper limit specified in requirement 1.5.5 should be tested. For this purpose, *LA51SAME.KB* is loaded into a plain text editor, one occurrence of

same is deleted, and the resulting file is saved under the name *LA50SAME.KB*. *ESIE*[™] loads this knowledge base without any problems.

This triggers the idea of testing the lower bound of *LEGALANSWERS*. Again, *LA51SAME.KB* is loaded into a plain text editor. All but one occurrence of *same* is deleted. The resulting knowledge base is saved as *LA1.KB* (Listing 133).

```

1 goal is something
2
3 legalanswers are same *
4
5 answer is "The answer is " something
6
7 question something is "Please, enter a value for something "
```

Listing 133: *ec/LA1.KB*

ESIE[™] loads this knowledge base without complaining. After starting a consultation with *GO* at the top level, the question is asked. The only valid answer the user can enter is *same*. This behavior is not useful.

The final test is to create a knowledge base called *LA0.KB*. Here *LEGALANSWERS* has no value, it terminates immediately with the asterisk, * (Listing 134).

```

1 goal is something
2
3 legalanswers are *
4
5 answer is "The answer is " something
6
7 question something is "Please, enter a value for something "
```

Listing 134: *ec/LA0.KB*

ESIE[™] loads this knowledge base without any problems. In the consultation, the question is asked, but the user cannot give a valid answer. The consequence is that *ESIE*[™] starts an endless loop in which the user is constantly prompted, *Please enter a value for something*. It is not possible to exit the program normally. It must be interrupted by pressing the *Ctrl* and *C* keys simultaneously, i.e. *Ctrl-C*. This aborts the program and returns control to the console window. This behavior is definitely not useful.

It is highly advisable to record these findings as additional requirements (Table 23). Now it becomes clear why the last number in the numbering scheme for requirements was increased in steps of five.

Normally, Table 23 would have to be integrated into Table 22 to create a new document with the consolidated requirements, which is not shown here.

ID	Topic	Subtopic	Description	Source
1.5.7	Syntax	LEGALANSWERS	LEGALANSWERS may have 0 or 1 <value>s, although this is not meaningful.	<i>ESIE</i> [™] , LA0.KB, LA1.KB

1.5.8	Syntax	LEGALANSWERS	All <value>s for LEGALANSWERS are stored without checking for duplicates.	ESIE™, LA50SAME.KB
2.1.16	User Interface	Start	If <value>s of LEGALANSWERS exceeds the maximum, the error message " Too many legalanswers encountered in the LEGALANSWERS rule." is issued.. → Syntax	ESIE™, LA51SAME.KB

Table 23: Further requirements for LEGALANSWERS

The question of which container template to use for storing the values for *LEGALANSWERS* has yet to be answered. This will be done now. To follow up on the arguments, <http://cppreference.com> and <http://www.cplusplus.com> are excellent sources. It must be pointed out again that the websites are recommended without guarantee. Caution is always advised when surfing the Internet, especially when downloading documents or programs.

As already indicated, `std::set<>` is not an appropriate choice. If the same element is inserted multiple times into a `set<>`, e.g. `set<string> values; values.insert("same"); values.insert("same");` it subsequently contains only one element "same". This is due to the property of a mathematical set, the archetype for `set<>`, which allows an element to occur only once. Unlike a mathematical set, `set<>` stores its elements in an internal order. As a result, the order in which the elements of a `set<>` are accessed is usually different from the order in which they were inserted. Of course, if the behavior of *ESIE™* is not to be mimicked exactly, `set<>` would be a good choice as a container template for storing *LEGALANSWERS* values.

To emulate the behavior of *ESIE™*, the `std::vector<>` container template is a better choice. A `vector<>` is a collection of elements that are ordered externally. That is, the elements in a `vector<>` are arranged in the order in which the programmer adds them to the `vector<>`. This ordering is preserved as long as no modifying algorithm is applied to the `vector<>`, such as sorting. Methods for adding elements are

- `vector<>::push_back()` – the element is added to the end of the vector, and
- `vector<>::insert()` – the first argument specifies an iterator and the second argument the element to be inserted before the iterator; elements to the right of the iterator are moved one position to the right.

`vector<>` does not care about duplicate elements. So the only attribute of `LegalAnswers` is `vector<string> m_answers;`. If a class consists of only one attribute, it is questionable whether it should be a class at all. According to the previous discussion, this is different for `LegalAnswers`, since important behaviors, i.e. methods, are added later. Figure 46 shows the – currently – simple class diagram.

LegalAnswers
m_answers : vector<string>

Figure 46: *LegalAnswers* class from the data perspective

6.4.1.4. Questions

Now it is discussed which container template is suitable for managing the individual questions in the Questions class. According to requirement 1.7.0, a question associates the name of a variable with a corresponding text. This suggests using the name of the variable as a key to access the text of the question. For this purpose, `std::map<>` is an optimal container template. A `map<>` does not allow duplicate keys and manages its key-value pairs in an internal order. The experience gained with *LEGALANSWERS* advises to create a special knowledge base to test whether or not questions with duplicate variable names are allowed by *ESIE™*.

The program shown in Listing 135 creates a valid knowledge base with 101 questions that all have the same variable name.

```

1 // 101SameQuestions.cpp by Ulrich Eisenecker, July 28, 2021
2
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     cout << "goal is something" << endl << endl;
9
10    // no legalanswers
11    // no rules
12
13
14    for (auto i { 1 }; i <= 101; ++i)
15    {
16        cout << "question same_variable"
17             << " is \"text\" << i << "\"\"
18             << endl << endl;
19    }
20
21    cout << "answer is \"The answer is \" something\" << endl << endl;
22 }
```

Listing 135: *ec/101SameQuestions.cpp*

After the program is compiled into *gen_kb*, it is executed as follows:

```
./gen_kb > 101SQ.KB
```

When loading this knowledge base, *ESIE™* gives the following error message and terminates:

There are too many questions in the Knowledge Base for me.

After deleting a question and saving the knowledge base as *100SQ.KB*, *ESIE™* loads it without complaint. If a syntactically valid knowledge base is created with multiple questions for the same variable, *ESIE™* will always ask the first question for that variable that occurs in the knowledge base. All other questions for that variable are ignored.

Thus, to accurately mimic the behavior of *ESIE™*, it is not possible to use `std::map<>`. As in the case of *LegalAnswers*, a `std::vector<>` is used to store each question. But there is a problem. A question consists of two components, namely the name of the variable and a text, but a `vector<>` can only manage single elements. Therefore, the `pair<>` template is used to group the components of a question. Thus, the declaration `vector<pair<string,string>> m_questions;` is added as a data member to the *Questions* class. Methods for loading a knowledge base and processing it will be added later. Figure 47 shows the corresponding class diagram.

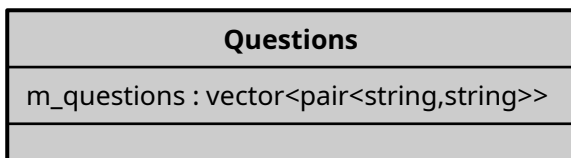


Figure 47: *Questions* class from the data perspective

These findings are also recorded as additional requirements (Table 24).

ID	Topic	Subtopic	Description	Source
1.7.6	Syntax	QUESTION	All QUESTIONs are recorded without checking if there are multiple questions for the same variable.	<i>ESIE™</i> , 100SQ.KB
1.7.7	Syntax	QUESTION	If there are multiple QUESTIONs for the same variable, all questions are asked in the order they appear in the knowledge base; each time, a question is asked, it can be answered differently.	<i>ESIE™</i> , KB not included
2.1.17	User Interface	Start	If the number of QUESTIONs exceeds the maximum, the error message " There are too many questions in the Knowledge Base for me." is output. → Syntax	<i>ESIE™</i> , 101SQ.KB

Table 24: Further requirements for *QUESTION*

6.4.1.5. Rule

Previously, it was decided that *KnowledgeBase* has a container for managing rules as a data member and to implement *Rule* as its own class. Before selecting a suitable container template, the *Rule* class is first designed.

A *Rule* consists of two parts. The first part consists of one or more *conditions* (in *MAN* they are called *comparisons*) and its second part is the *conclusion*. Each condition and each conclusion count as one rule line (requirement 1.6.5). A single condition consists of the name of a variable and a value. Both parts belong closely together. Therefore they are represented as a `pair<string,string>`. So far, there is no requirement specifying the order in which the conditions of a rule are stored.

The knowledge base shown in Listing 136 helps clarify the order in which conditions are stored in a rule.

```
1 goal is something
2
3 legalanswers are yes no *
4
5 answer is "The answer is " something
6
7 if a is yes
8 and b is yes
9 then something is ab
10
11 question a is "a is "
12
13 question b is "b is "
```

Listing 136: *ec/ORDERCND.KB*

If this knowledge base is loaded and a consultation is started, the question for *a* is asked first and then the question for *b* is asked. If the rule is changed as shown in Listing 137, the question about *b* is asked first and then the question about *a*.

```
1 if b is yes
2 and a is yes
3 then something is ab
```

Listing 137: *Rule of ec/ORDERCND.KB with reverse order of conditions*

This is not a proof, but a clear indication that the conditions are stored in the order in which they normally appear in the knowledge base.

The next question is whether *ESIE*[™] checks for duplicate conditions. To find out, *ec/ORDERCND.KB* is loaded and *TRACE ON* is entered before the consultation is started. *ESIE*[™] then outputs the information shown in Figure 48.

```
There were 3 rule-lines, 2 questions and 2
legal answers specified in the knowledge base.
```

Figure 48: *Output for consulting ec/ORDERCND.KB with tracing enabled*

Now duplicate conditions are introduced into the rule of *ec/ORDERCND.KB*. Listing 138 shows the resulting knowledge base.

```
1 goal is something
2
3 legalanswers are yes no *
4
5 answer is "The answer is " something
6
7 if a is yes
8 and b is yes
9 and b is yes
10 then something is ab
11
12 question a is "a is "
13
14 question b is "b is "
```

Listing 138: *ec/DUPLCND.KB*

After loading this knowledge base and entering *TRACE ON*, the *ESIE™* consultation provides the result shown in Figure 49.

```
There were 5 rule-lines, 2 questions and 2
legal answers specified in the knowledge base.
```

Figure 49: Output for consulting *ec/DUPLCND.KB* with tracing enabled

Obviously, *ESIE™* does not check for duplicate conditions. At the beginning of the consultation, the questions for *a* and *b* are asked only once. The duplicates are not a problem, but they consume rule lines.

The next question is: How are contradictory conditions handled? The knowledge base shown in Listing 139 is based on *ec/ORDERCND.KB*, but introduces a contradiction in its only rule.

```
1 goal is something
2
3 legalanswers are yes no *
4
5 answer is "The answer is " something
6
7 if a is yes
8 and b is yes
9 and b is no
10 then something is ab
11
12 question a is "a is "
13
14 question b is "b is "
```

Listing 139: *ec/CONTRCND.KB*

It is impossible for *b* to be *yes* and *no* at the same time. Nevertheless, *ESIE™* loads this knowledge base without complaining. After starting a consultation, it asks the question for *a*, then the question for *b*. If the first question is answered *yes*, it does not matter how the second question is answered. In both cases, *ESIE™* reports an error in the knowledge base (Figure 50). If the first question is answered *no*, no further question is asked and the error shown in Figure 50 is reported immediately.

```
Error in Knowledge Base.
SOMETHING searched for but not found.
```

Figure 50: Error message when consulting *ec/CONTRCND.KB*

This is an excellent example of an error that could in principle be detected during *load time* (an equivalent of *compile time*), but is not. Rather, its consequences cause an error at *consultation time* (a *run-time* equivalent). Normally, one would strive to detect and report as many errors of this type as early as possible, i.e., during load time. But *ESIE™* does not do this. This (erroneous) behavior must be recorded as an additional requirement, which will be done at the end of the Rules Section.

Based on this extensive research, `std::vector<>` appears to be a suitable container template for storing conditions. Previously, it was pointed out that a single condition

is represented as a `pair<string,string>`. Therefore, the data member `vector<pair<string,string>> m_conditions;` is added to the Rule class. The conclusion combines the name of a variable and a value assigned to the variable when the rule is *fired*, that is, when all its conditions evaluate to true. Since the variable name must be accessed when checking which rule to evaluate, both the variable name and the value are represented directly as members of type `string`, namely `string m_variable, m_value;`. This results in the class diagram for Rule shown in Figure 51.

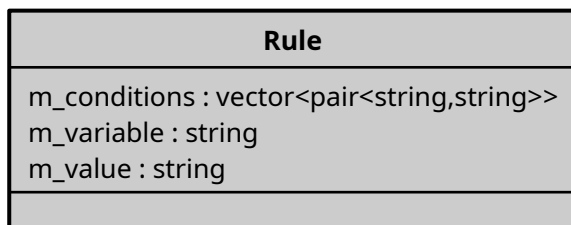


Figure 51: Rule class from the data perspective

6.4.1.6. Rules

Now for the Rules class. Based on the given knowledge, it can be assumed that *ESIE™* does not check for duplicate rules, neither literal nor structural, but stores them anyway. *Literal* means that two rules are identical in terms of their conditions and order, as well as the conclusion. *Structural* means that two rules are identical in terms of their conditions, but not necessarily in terms of their order, as well as the conclusion.

First, it is examined what happens when the knowledge base has 401 rule lines. This is relevant because the error message that *ESIE™* issues in this case is not documented anywhere. The program in Listing 140 generates a corresponding knowledge base with 199 literally identical rules plus one rule with three rule lines. The generation of these rules by a program makes sense, since the manual creation of such a knowledge base would be tedious and error-prone.

```

1 // 401RuleLines.cpp by Ulrich Eisenecker, July 29, 2021
2
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     cout << "goal is something" << endl << endl;
9
10    // no legalanswers
11
12    for (auto i { 1 }; i <= 199; ++i)
13    {
14        cout << "if a is yes\nthen something is true\n" << endl;
15    }
16    cout << "if a is yes\nand b is yes\nthen something is true\n" << endl;

```



```

17
18 cout << "answer is \"The answer is \" something" << endl << endl;
19
20 cout << "question a is \"value of a\"\\n" << endl;
21 cout << "question b is \"value of b\"\\n" << endl;
22 }

```

Listing 140: `ec/401RuleLines.cpp`

Compiling this program to `gen_kb` and then running it by entering `./gen_kb > 401RL.KB` creates the `401RL.KB` knowledge base. When loading this knowledge base, *ESIE™* reports the error “*There are too many rules in the Knowledge Base for me.*” and terminates.

Now the first rule in `401RL.KB` is deleted with a simple text editor and the result is saved as `198DUPRL.KB`. The reason for the name of the file is that there are 198 literally identical rules and one other rule. Doing this as a manual step makes sense because there is only little chance of error in this manual step. Doing it manually is also more economical than writing a program.

ESIE™ loads `198DUPRL.KB` without any problems. After entering `TRACE ON`, *ESIE™* correctly reports that the knowledge base contains 399 rule lines, 2 questions and 0 legal answers. The consultation runs without any problems. When the first question for *a* is answered *yes*, *ESIE™* reports “*The answer is TRUE*” and ends the consultation. It is a new additional requirement that *ESIE™* does not check for identical rules, but stores them all, consuming rule lines. Duplicate rules do not cause problems during a consultation.

Overall, this argues for `std::vector<>` as a container template for storing rules. Consequently, `vector<Rule> m_rules;` is the only member of the `Rules` class. Other behavior, for example adding a rule and processing rules, will be added later. Therefore, it makes sense to design `Rules` as its own class, as shown in Figure 52.

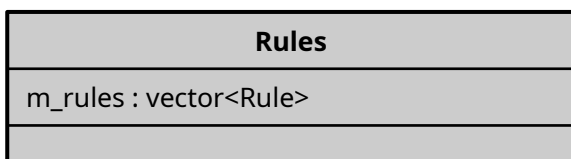


Figure 52: *Rules class from the data perspective*

Table 25 contains the requirements found in the closer examination of *ESIE™* in the previous and the current section. As already written, they would have to be integrated into the requirements document, which is not done here since repeating the ever-growing table of requirements would not make didactic sense.

ID	Topic	Subtopic	Description	Source
1.6.1	Syntax	IF	The conditions are stored in the order in which they appear in a rule in the knowledge base.	<i>ESIE™</i> , ORDERCND.KB
1.6.6	Syntax	IF	<i>ESIE™</i> does not check for duplicate conditions. Consequently	<i>ESIE™</i> ,

ID	Topic	Subtopic	Description	Source
			they are all stored. The duplicates do not seem to cause any problems during a consultation.	DUPLCND.KB
1.6.7	Syntax	IF	<i>ESIE™</i> does not check for duplicate rules. Consequently they are all stored. The duplicates seem to cause no problem during a consultation.	<i>ESIE™</i> , 198DUPRL.KB
2.2.21	User Interface	Top Level	After entering TRACE ON, the following information is reported: "There were ?? rule-lines, ?? questions and ??\n legal answers specified in the knowledge base.", where \n stands for a line feed and ?? for the actual numbers.	<i>ESIE™</i> , CONTRCND.KB
3.2.1	Processing	Load Knowledge Base	If more than 400 rule lines are detected (→ requirements 1.6.5 and 1.6.10), the following error message is displayed: "There are too many rules in the Knowledge Base for me."	<i>ESIE™</i> , 401RL.KB
3.3.6	Processing	Consultation	If an error is found in the knowledge base logic, the error message is "Error in Knowledge Base.\n ?? searched for but not found.", where \n stands for a line feed and ?? for the name of the searched element).	<i>ESIE™</i> , CONTRCND.KB

Table 25: Further requirements for RULE and RULES

6.4.1.7. Variables

Previously, it was explained that a container is required for storing variable names and values. Therefore, the `Variables m_variables;` data member was added to the `KnowledgeBase` class. The decisions about the container template to manage the variables and the design of the concept *variable* were postponed. This now needs to be decided. The first question is whether the concept *variable* is its own class. To answer this question, the properties of the concept *variable* are examined. A *variable* has a *name* by which it is referred to and a *value*. Since two variables cannot have the same name, the name must be unique. However, a single variable cannot judge whether its name is unique or not. Only the container that manages all variables can do this. The value of a variable can be set or not. It is usually important to know whether a variable has already been initialized with a value or not. In *ESIE™*, variables only have string values, and a variable with an empty string as its value is considered uninitialized. That is, an empty string is not part of the range of valid values. Therefore, the value of a variable in this case can serve two purposes:

- A value with an empty string means that the variable is not initialized,
- any other value means that the variable is initialized with exactly this value.

Due to the peculiarities of variable processing in *ESIE™*, it should not happen that a variable is assigned more than once. *ESIE™* only attempts to assign values to variables that have not yet been initialized. So there is no need to check for multiple assignments. These are all arguments against designing a variable as its own class. Instead, variable names and their values can be managed by the container.

Since the value of a variable is accessed by its name, a `std::map<>` would be a suitable container template. It stores key/value pairs and allows accessing a value,

i.e. the value of a variable, by its key, i.e. the name of a variable. This is very convenient and efficient. In addition, `map<>` automatically prevents duplicate entries. On the other hand, a `map<>` stores its entries, the key/value pairs, in its own internal order. Thus, the order of the variables as they occur in questions, rules, target, and answer may not be preserved. To avoid possible problems due to a different order in principle, `std::vector<>` is again chosen to manage the variables. Please note that this is purely a precautionary measure that can be checked later when the program has been completed and executed without errors. Since a *variable* consists of a *name* and a *value*, both of type `std::string`, `vector<>` is instantiated with `pair<string, string>`. Let `element` be an element of the container, `element.first` is the *name* of the variable and `element.second` its *value*. Consequently, the data member `vector<pair<string, string>> m_variables;` is added to the `Variables` class. Later, when the application compiles and runs correctly, it can be checked if `std::vector<>` can be safely replaced by `std::map<>`.

That `Variables` is a standalone class is justified because some methods are added later. It also ensures that there are no duplicates of variables in the `Variables::m_variables` container. Figure 53 shows the class diagram for `Variables` from the data perspective.

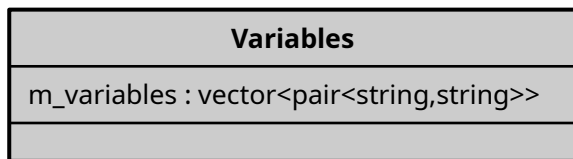


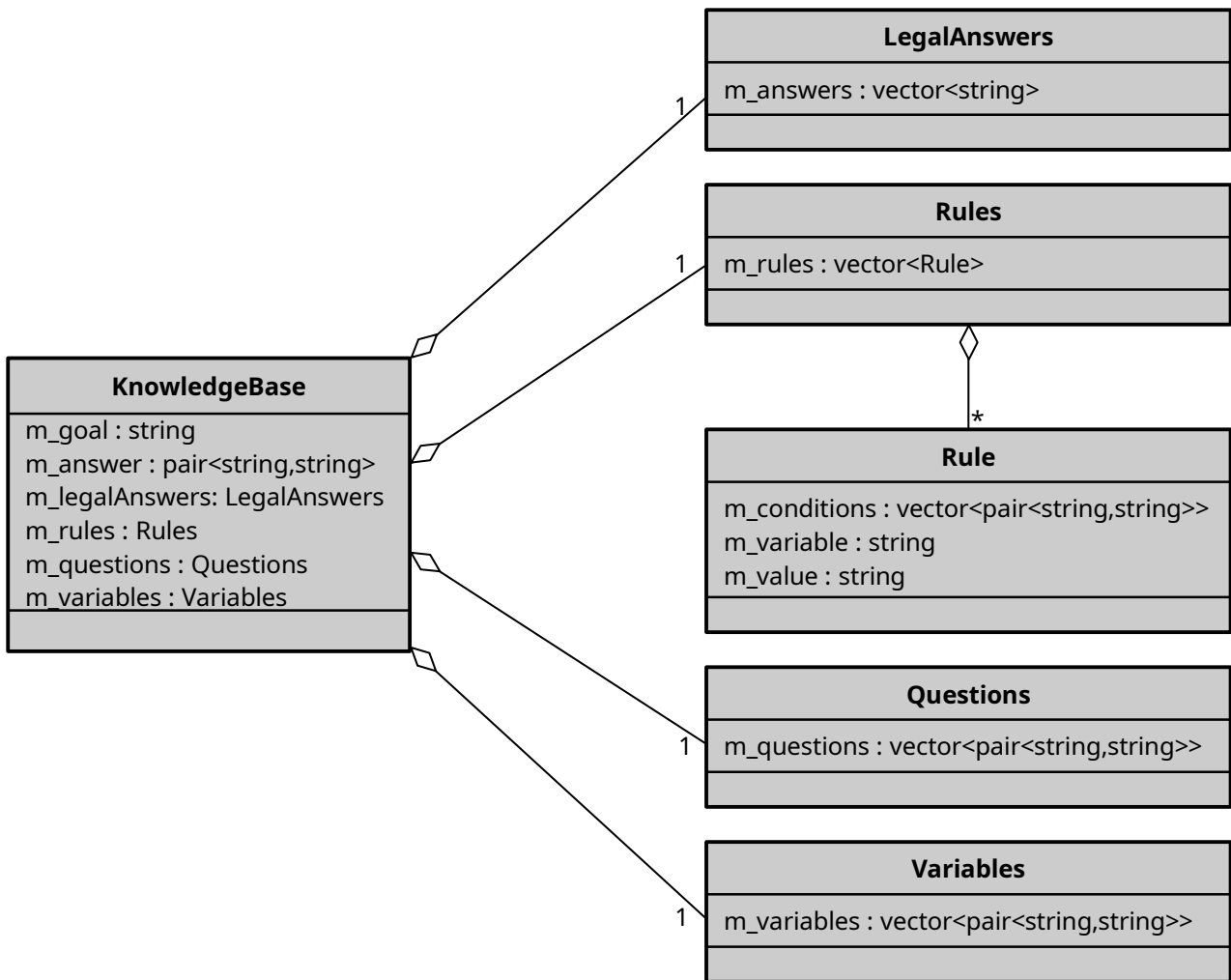
Figure 53: *Variables class from the data perspective*

The methods have been intentionally omitted. They will be included again in the design of loading and processing. The representation of aggregation relationships is redundant since all classes have appropriate attributes for managing exemplars of the aggregated classes. Some classes have disappeared because they have been integrated into other classes. For example, *Goal* and *Answer* from the analysis model are now attributes of `KnowledgeBase`. It is very interesting to see how different the class diagrams from analysis and data design are. None of them is useless because all class diagrams represent the system under development from different points of view. Also, all diagrams should be updated as new requirements emerge or progress is made in design and implementation.

6.4.1.8. Data Perspective Consolidated

Figure 54 shows the revised class diagram for the classes and their relationships from a data design perspective.

Figure 54: *Consolidated class diagram from the data perspective*



6.4.1.9. Critical Review and Completion

Now it is time to critically review the work done so far to design the system from a data perspective!

The classes have been thoroughly revised and attributes for aggregations have been added. New requirements have been found, especially with respect to certain limits, associated error messages, and possible error states and their handling. The following questions are still open:

1. What happens if a variable name exceeds 40 characters (requirement 1.2.35)?
2. What happens if a value exceeds 40 characters (requirement 1.2.35)?
3. What happens if the text of a question exceeds 80 characters (requirement 1.2.45)?

Of course, when writing this text, it would have been easy to revise it so that these issues were addressed earlier and in a more appropriate context. But they have been mentioned here on purpose. When validating and adding requirements, humans rarely work perfectly. This must be taken into account when developing methods and tools. One principle followed here is the iterative flow of requirement

analysis, analysis modeling, design modeling, and – later – implementation. This iterative and incremental approach helps to uncover open issues and omissions, as shown here.

The knowledge base shown in Listing 141 has a variable with a 41 character name.

```
1 goal is 123456789a123456789b123456789c123456789d1
2
3 answer is "The answer
4 is " 123456789a123456789b123456789c123456789d1
5
6 question 123456789a123456789b123456789c123456789d1
7 is "Please, enter a value for something "
```

Listing 141: *ec/VAR41.KB*

ESIE™ successfully loads this knowledge base without reporting an error. *TRACE ON* is then entered at the top level to have *ESIE™* output the names of the variables it is currently searching for. After *GO* is entered, *ESIE™* reports that it is currently searching for *123456789A123456789B123456789C123456789D*. Apparently, a variable name longer than 40 characters is silently truncated. As a result, *ESIE™* cannot distinguish between variables whose names are different from character position 41 upwards. In addition, the lowercase letters in the knowledge base have been replaced with uppercase letters. This is a clear indication that lowercase letters in variable names are replaced with uppercase letters before they are stored to comply with requirement 1.1.0.

The knowledge base shown in Listing 142 contains two values longer than 40 characters.

```
1 goal is something
2
3 answer is "The answer is "
4 something
5
6 if a is yes
7 then b
8 is 123456789a123456789b123456789c123456789d1
9
10 if b is 123456789a123456789b123456789c123456789d1
11 then something
12 is abcdefghi1abcdefghijklm2abcdefghijklm3abcdefghijklm4a
13
14 question a
15 is "value of a"
```

Listing 142: *ec/VAL41.KB*

When loading this knowledge base, *ESIE™* does not report any warnings or errors. Before the consultation begins, *TRACE ON* is entered. After the question *value of a* is answered *yes*, *ESIE™* outputs that it has learned that *B* is *123456789A123456789B123456789C123456789D* and that *SOMETHING* is *ABCDEFGHI1ABCDEFGHI2ABCDEFGHI3ABCDEFGHI*. This shows that values are treated in the same way as variable names. If their length exceeds 40 characters,

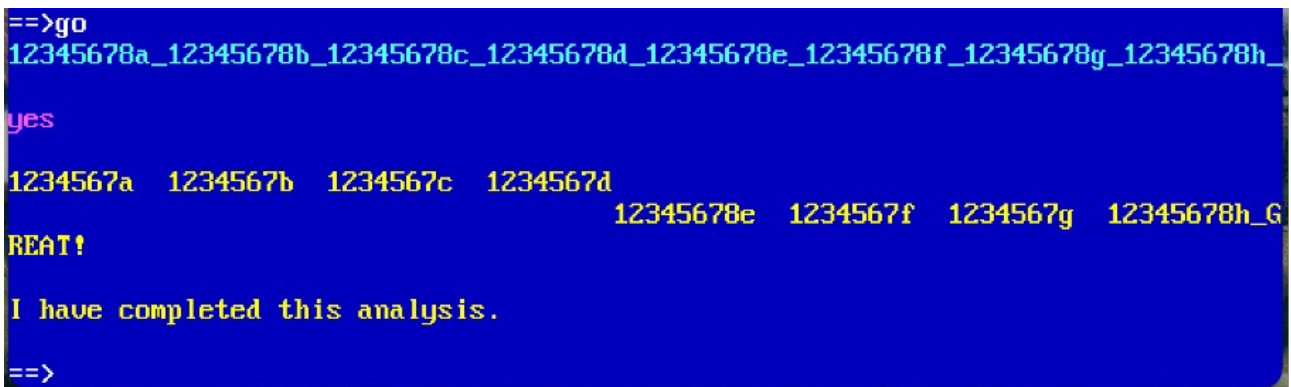
they are silently truncated to 40 characters. Also, the values are stored with all lowercase letters previously converted to uppercase.

Last but not least, it is examined how *ESIE*[™] handles text in answers and questions that is longer than 80 characters. The knowledge base shown in Listing 143 contains two texts, each 81 characters long. One of them consists of consecutive non-whitespace characters, the other contains additional whitespace characters.

```
1 goal is something
2
3 answer is
4 "1234567a 1234567b 1234567c 1234567d
5 12345678e 12345678f 12345678g 12345678h_1"
6 something
7
8 if a is yes
9 then something is great!
10
11 question a
12 is "12345678a_12345678b_12345678c_12345678d_12345678e_12345678f_12345678g_12345678h_1"
```

Listing 143: *ec/TXT81.KB*

ESIE[™] loads the knowledge base without complaint. After entering *GO* at the top level, *ESIE*[™] asks the question. The text of the question is truncated after the last underscore, i.e. after 80 characters. If one answers the question with *yes*, one gets an interesting result, as shown in the screenshot in Figure 55.



The screenshot shows a terminal window with a blue background. The prompt is ==>. The first line of the question is truncated to 80 characters: 12345678a_12345678b_12345678c_12345678d_12345678e_12345678f_12345678g_12345678h_. The user responds with 'yes' in pink. The answer is displayed in yellow: 1234567a 1234567b 1234567c 1234567d followed by a line feed character, then 12345678e 1234567f 1234567g 12345678h_GREAT!. The final line of the answer is 'I have completed this analysis.' in yellow. The prompt ==> is visible at the bottom.

Figure 55: *Processing of ec/TXT81.KB saved in UNIX format*

All whitespaces in the response are retained, including the newline character. After *123467d* there are no more whitespaces, which has been checked with a text editor. The line feed character, which is also a whitespace, causes a change to the next line where the output continues. The text is truncated after 80 characters.

The knowledge base was stored in *Unix format*, with lines terminated with a single line feed. *ESIE*[™] is an *MS-DOS* application and therefore expects text files in *MS-DOS format*. Under *MS-DOS*, a line of text is terminated with a carriage return/line feed sequence. After saving the knowledge base in *MS-DOS format* with the name *ec/TXT81DOS.KB*, the output changes and looks as expected (Figure 56).

```
==>go
12345678a_12345678b_12345678c_12345678d_12345678e_12345678f_12345678g_12345678h
yes
1234567a 1234567b 1234567c 1234567d
12345678e 12345678f 12345678g 12345678h_GREAT!
I have completed this analysis.
==>_
```

Figure 56: Processing of *ec/TXT81DOS.KB* saved in MS-DOS format

Out of curiosity, the knowledge base is saved in *MAC format* with the name *TXT81MAC.KB*. Under *macOS*, text lines are terminated only with a carriage return.

After loading the knowledge base, starting the consultation and entering the answer *yes*, another output is generated as shown in Figure 57.

```
==>go
12345678a_12345678b_12345678c_12345678d_12345678e_12345678f_12345678g_12345678h
yes
12345678e 12345678f 12345678g 12345678h_1GREAT!
I have completed this analysis.
==>
```

Figure 57: Processing of *ec/TXT81MAC.KB* saved in macOS format

Now the first line of the answer text is missing. This is because the output of the first line ends with a carriage return. This moves the output cursor to the first column where the second line of text overwrites the first line.

It should be noted that lowercase letters are preserved in the text of answers and questions, i.e. they are not converted to uppercase. Therefore, requirement 1.1.0 must be adapted accordingly. In addition, requirement 1.2.40 is correct, but it must be emphasized that non-printable control characters are also preserved. In addition, it is expressly pointed out that it is not possible to insert a double quotation mark in a text, as a double quotation mark either begins or ends a text.

It is assumed that invalid commands were unknowingly entered at the top level during the previous investigations. This showed that *ESIE™* reliably detects invalid commands at the top level and reports them as errors.

Table 26 shows the changed and added requirements. For changed requirements, the text of the previous version is formatted in *italics and highlighted in light blue*.

ID	Topic	Subtopic	Description	Source
1.1.0	Syntax	General	With the exception of text in variables and questions, <i>the program is case insensitive everywhere.</i>	MAN, p.13, ESIE™, TEXT81.KB
1.2.36	Syntax	Knowledge Base	Variable names and values longer than 40 characters are silently truncated to a length of 40 characters.	ESIE™, VAR41.KB, VAL41.KB
1.3.37	Syntax	Knowledge Base	Lowercase letters in variable names and values are replaced by corresponding uppercase letters before saving.	ESIE™, VAR41.KB, VAL41.KB
1.2.40	Syntax	Knowledge Base	<i>A <text> part is enclosed in double quotes "", and may contain any of the 255 characters of the extended ASCII character set except the double quote, including non-printable control characters. It is not possible to include a double quotation mark in <text>.</i>	MAN, p. 15, ESIE™, TEXT81.KB, TXT81DOS.KB, TXT81-MAC.KB
1.2.46	Syntax	Knowledge Base	Text that longer than 80 characters is silently truncated to 80 characters.	ESIE™, TXT81DOS.KB, TXTM81MAC.KB
2.2.6	User Interface	Top Level	When entering an invalid top-level command, the error message "I don't understand that command.\n\nValid options are: TRACE ON, TRACE OFF, GO, AND EIT." is issued (\n stands for a line feed).	ESIE™, any KB

Table 26: Further requirements for variable names, text and Top Level

This concludes the design from a data perspective for now. Next, the design will be refined with regard to loading a knowledge base.

6.4.2. Loading

According to the captured requirements, a knowledge base is a simple text file with five rule types that can occur in any order. Each of the rule types consists of character strings. There are two categories of strings: any sequence of characters without spaces and any sequence of characters enclosed in double quotation marks. For loading a knowledge base, it is necessary to explore how strings can be entered in C++. The program shown in the Listing 144 helps to understand how C++ inputs strings.

```

1 // InputToken.cpp by Ulrich Eisenecker, January 23,2023
2
3 #include <iostream>
4 #include <string>
5
6 using namespace std;
7
8 int main()
9 {
10     string s;
11     while (cin)
12     {
13         cin >> s;
14         cout << s << endl;
15     }
16 }

```


Listing 144: *ec/InputToken.cpp*

Figure 58 shows how to compile the program and run it in a terminal window. It also shows some lines of the generated output file. The user's input is shown in a **blue bold font**.

```
ec % g++ -o it InputToken.cpp
ec % ./it < UN_AD_CN.KB > output.txt
ec % cat output.txt
goal
is
container
legalanswers
are
yes
no
*
answer
is
"The
STL
container
matching
your
needs
is
probably
a(n)
"
```

Figure 58: *Processing of inputs by the ec/InputToken.cpp program*

The prompt indicates that *ec* is the current directory. This is important because *ec* contains the knowledge base *UN_AD_CN.KB*.

The command in the first line compiles *InputToken.cpp* into the executable file *it* specified after the *-o* option. It is then executed. To execute a binary file in a UNIX-based operating system, its name must be preceded by *./*. The dot represents the current directory and */* separates the directory from the file name. The first chevron, *<*, redirects the program's input from the terminal window to *UN_AD_CN.KB*. The second chevron, *>*, redirects the program's output to the *output.txt* file. Then the *cat* command is used to display *output.txt*.

Apparently rule names, i.e. *goal*, *legalanswers*, *answer*, etc., reserved words and special characters, e.g. *is* and ***, as well as variable names and values are read as required. After extraction, lowercase letters need to be converted to uppercase. The situation is different for strings in double quotation marks.

Of course, the program can also be started interactively to see how it accepts strings in double quotes. Figure 59 shows an example dialog. Again, the user's input is displayed in **blue letters**.

The user input contains several spaces and tabs. To signal the end of the input stream, an *end-of-file* character must be entered, e.g. *Ctrl-D* on *macOS* or *Ctrl-Z* on *MS Windows*. Alternatively, the program can be interrupted by pressing *Ctrl-C* (on both *macOS* and *MS Windows*). When a string is extracted with the `>>` input operator (also known as the *stream extraction operator*), all whitespace characters at the beginning of the string are consumed and discarded. After that, all non-whitespace characters are read and preserved until a whitespace character occurs. The whitespace character is not extracted and remains in the input stream; the preserved characters are returned as the resulting string. One way to extract a quoted string is to input tokens, concatenate them until the initial quote character matches a closing quote character, and return the resulting string without the double quotation marks. Unfortunately, this would remove all whitespaces.

```
"How are you?"
"How
are
you?"
           "How  are           you?  "
"How
are
you?"
"
```

Figure 59: Interactive execution of the `ec/InputToken.cpp` program

Therefore, this approach is completely abandoned. Instead, a function is developed that inputs either strings without double quotation marks, i.e., strings that contain neither spaces nor double quotation marks, or strings enclosed in double quotation marks but do not contain them.

The program in Listing 145 shows a prototypical implementation of the `inputToken()` function that does just that. `inputToken()` accepts a reference to an `istream`. Thus, it accepts input file streams, input string streams, or any other type-compatible class. It returns the extracted token as a value of type `std::string`.

```
1 // Input2TokenTypes.cpp Ulrich Eisenecker, January 24, 2023
2
3 #include <iostream>
4 #include <string>
5 #include <cctype> // because of toupper()
6 using namespace std;
7
8 [[nodiscard]] string inputToken(istream& is)
9 {
10     string token { };
11     is >> ws; // remove leading whitespace
12     if (is.peek() == '\\')
13     {
14         is.get(); // discard initial quote
15         while (is && is.peek() != '\\')
16         {
17             token += is.get();
18         }
19         if (is && is.peek() == '\\')
20         {
```

```

21         is.get(); // discard closing quote
22     }
23 }
24 else
25 {
26     while (is && is.peek() > 32 && is.peek() != '\")
27     {
28         token += toupper(is.get());
29     }
30 };
31 return token;
32 }
33
34 int main()
35 {
36     string token { };
37     while (cin)
38     {
39         token = inputToken(cin);
40         if (cin)
41         {
42             cout << '[' << token << ']' << endl;
43         }
44     }
45 }

```

Listing 145: `ec/Input2TokenTypes.cpp`

The first statement, `is >> ws;` (line 11), removes leading whitespace by extracting the `std::ws` manipulator from the input stream `is`. After that, the next character in `is` is either a non-whitespace character or the end of the file has been reached. In the latter case, all subsequent input operations are ignored.

In the first case, the condition of the `if` statement is executed (line 12). The `istream::peek()` member function searches for the next character in the input stream, but does not extract it. If no character is currently available because `istream::peek()` is executing on `cin`, it waits until a character becomes available. If the next character is a double quote, the then part of the `if` statement is executed, which extracts a string enclosed in a pair of double quotes (lines 13 through 23). Otherwise, the `else` part is executed, which extracts a string that does not contain spaces or double quotes (lines 25 through 30).

`istream::get()` (line 15), an overloaded member function of `istream`, extracts exactly one character and returns its character code value as `int`. When the end of the input stream is reached, `istream::get()` returns the *end-of-file* (EOF) value. Here, the result of the function call is ignored, because the only purpose of the function call is to extract and discard the double quotation mark. The next statement is a `while` loop (line 15). Its condition depends on the short-circuit evaluation of logical expressions in C++ (Section [bool Type](#)). The first part of the condition, `is`, evaluates to `true` if no error flag is set for the input stream (line 15). This is possible because there is a type conversion operator defined for streams that converts a stream to a value of type `bool` with respect to the current state of the stream, i.e. `true` if the stream is good and `false` otherwise. If the value is `false`, the

second part of the condition, `is.peek() != '\\"'`, is not evaluated at all. If the value is true, `is.peek() != '\\"'` is evaluated. If the next character is not a double quotation mark, it is extracted and appended to the token, namely: `token += is.get();` (line 17), otherwise nothing is done. When the while loop is finished, the if statement extracts the terminating double quotation mark if the stream is still valid (lines 19 – 22).

The else branch of the top-level if statement (lines 25 through 30) extracts a string that does not contain spaces or double quotes. The else branch consists of a single while loop with a complex condition that also exploits the short-circuit evaluation of logical expressions (line 26). The first part checks if the stream is still OK. If it is, the second part is evaluated. It checks if `is.peek()` returns a character with a value greater than 32. 32 is the ASCII value of the *space character*. All characters with an ASCII value less than 32 are considered as whitespaces. If this condition is met, the third condition is tested. It is checked whether `is.peek()` is not a quote character. If this applies, the statement `token += toupper(is.get());` (line 28) is executed. As explained earlier, `is.get()` extracts the next character from the stream `is` and returns its ASCII value as `int`. Normally, `std::toupper()` accepts a value parameter of a `char`. Here, an automatic type conversion of the ASCII value to a value of type `char` is performed. If the passed value is a lowercase character, the value of the corresponding uppercase character is returned, otherwise the value is returned unchanged. Then, the value returned by `std::toupper()` is converted to `char` by an automatic type conversion and appended to the C++ string `token`. Finally, `token` is returned as the result.

The central part of the `main()` function is a while loop that is executed as long as `cin` is valid (lines 37ff.). It calls the `inputToken()` function and passes `cin` to it as a parameter. The result is assigned to the string `token` defined in `main()`. If `cin` is still valid after the last input operation, the extracted token is sent to `cout`, enclosed in square brackets. This is useful to judge whether all characters were extracted correctly. The additional check of `cin` before sending the just extracted token to the output is necessary because `cin` might have immediately contained an end-of-file character or only spaces followed by an end-of-file character.

After having compiled this program as an executable, it is highly recommended to try it out. Executing `./it < UN_AD_CN.KB` in a console window will also give an idea of how this program works.

Now that `inputToken()` is available, it is easy to think about loading a knowledge base.

First, `inputToken()` becomes a member function of the `KnowledgeBase` class. Second, a member function `input()` is added to `KnowledgeBase` because it is responsible for loading the entire knowledge base. Thus, `KnowledgeBase::input()` is the entry point for loading. It can be conceptualized as a while loop that executes

as long as the input stream containing the knowledge base is in a good state. It first calls `inputToken()` and then checks if the result matches an *ESIE*[™] keyword. If it does, another member function is called to input the rest of the corresponding rule.

Listing 146 gives an impression of the structure of this while loop. It uses so-called *pseudo code*, which resembles a programming language but is not intended to be interpreted or compiled. As a mixture of natural and formal language, it is used to outline an algorithm or a procedure to be design.

```
1 while (input is ok)
2 {
3     token = inputToken(input)
4     if token equals "LEGALANSWERS"
5         inputLegalAnswers(input)
6     else if token equals "GOAL"
7         inputGoal(input)
8     else if token equals "IF"
9         inputRule(input)
10    else if token equals "QUESTION"
11        inputQuestion(input)
12    else if token equals "ANSWER"
13        inputAnswer(input)
14    else if token is not empty
15        output Invalid Rule in Knowledge Base.
16 }
```

Listing 146: Pseudo code for loading a knowledge base

All methods mentioned here for the first time become member functions of `KnowledgeBase`. Again, pseudo code is used to outline what the more complex of these new member functions do. It must be emphasized that each new input method inputs the remaining syntactic elements of the corresponding entity, since its first element has already been extracted. `KnowledgeBase::inputLegalAnswers` (Listing 147) first calls the new method `KnowledgeBase::inputIsAre()`, which consumes the filler "IS" or "ARE". It then starts a do loop that calls `KnowledgeBase::inputToken()`. If the result is not the *splat* (asterisk, *), the `m_legalAnswers.add()` method is called to add the token as a new answer. Once the asterisk is extracted, the do loop is terminated. Listing 147 shows the structure of this do loop as pseudo code.

```
1 inputIsAre(input)
2 string answer;
3 do
4 {
5     answer = inputToken(input)
6     if answer isNot "*"
7     {
8         m_legalAnswers.add(answer)
9     }
10 } while answer isNot "*"
```

Listing 147: Pseudo code for inputting a LegalAnswer

As written before, `KnowledgeBase::inputIsAre()` reads the next token from the input stream, which should have the value "IS" or "ARE" according to the

requirements 1.4.0, 1.5.0, 1.6.0, 1.6.5 and 1.7.0. Since this method is relatively simple, no pseudo code is given for it.

Again, all emergent methods are captured as new member functions of the corresponding classes.

`KnowledgeBase::inputGoal()` first calls `KnowledgeBase::inputIsAre()`. Then, `m_goal` is assigned the result of the call to `KnowledgeBase::inputToken()`. Finally, `m_goal` is added to the attribute `m_variables` by calling `Variables::add()`. Because of the simplicity of this method, no pseudo code is shown for it.

`KnowledgeBase::inputRule()` (Listing 148) consists of two sections. The first section defines a local variable called `rule` of type `Rule`. The following do loop attempts to input a variable-value pair from the input stream. This is achieved by a new member function `KnowledgeBase::inputVariableValue()`. If the input is successful, the variable is added to `m_variables`, and the variable and value are added to the rule as a new condition by calling `rule.addCondition(variable,value);`. Then the next token is extracted from the input. The do loop is repeated as long as this token equals "AND". The second section starts, if the last extracted token has the value "THEN". In this case a variable-value pair is extracted again from the input stream. If successful, the variable is added to `m_variables`, and the conclusion is added to rule by calling `rule.addConclusion()`. This completes the rule input, and the rule is added to `m_rules` by calling `m_rules::add()`. Listing 148 shows the pseudo code for both sections of the member function `KnowledgeBase::inputRule()`.

```
1 string variable, value, token
2 Rule rule
3 do
4 {
5     inputVariableValue(input,variable,value)
6     m_variables.add(variable)
7     rule.addCondition(variable,value)
8     token = inputToken(input)
9 } while token equals "AND"
10
11 if token equals "THEN"
12 {
13     inputVariableValue(input,variable,value)
14     m_variables.add(variable)
15     rule.addConclusion(variable,value)
16     m_rules.add(rule)
17 }
```

Listing 148: Pseudo code for inputting a Rule

The member function `KnowledgeBase::inputVariableValue()` extracts the name of the variable, calls `KnowledgeBase::inputIsAre()`, and extracts a value from the input stream. This method is relatively simple, so no pseudo code is given for it.

The next input method to be discussed is `KnowledgeBase::inputQuestion()` (Listing 149). Its first action is to extract the name of the variable. After that, `KnowledgeBase::inputIsAre()` is called. Next, the subject, i.e. the text of the question, is extracted. Finally, the variable name is added to the variable store by

calling `m_variables.add(variable);`, and the `variable` and `subject` are added to the question store by calling `m_questions.add(variable,subject);`. Listing 149 shows the pseudo code for this method.

```
1 string variable
2 variable = inputToken(input)
3 inputIsAre(input)
4 string subject
5 subject = inputToken(input)
6 m_variables.add(variable)
7 m_questions.add(variable, subject)
```

Listing 149: Pseudo code for inputting a Question

`KnowledgeBase::inputAnswer()` completes the various input methods. First, it checks if an answer has already been read from the knowledge base. This is the case if `m_answer.first` or `m_answer.second` is not the empty string. Next, `KnowledgeBase::inputIsAre()` is called. Then the subject and the name of the variable are extracted. Again, the variable is added to the variable store, and finally the subject and variable are stored as a pair in `m_answer`. The structure of this method is similar to that of extracting a question. Therefore, no pseudo code is shown for this method.

The class diagram in Figure 60 now shows everything from the perspectives of data and loading.

6.4.3. Processing

Requirements 3.3.10, 3.3.15, 3.3.20, 3.3.50, and 3.3.70 state that the program manages a *stack* that is important for processing. The requirements describe some relevant aspects, such as that the *goal* is the first element pushed onto the *stack* and when an element is popped from the *stack*. However, they do not describe what else is put on the *stack* and when. This information seems insufficient to make the knowledge base processing accurate. How can this problem be fixed? Of course, one could contact the author of the software. Another highly recommended option is to get books on inference in rule-based expert systems and read them, or search the Internet for relevant information. In addition, one can take a closer look at the output with the trace function turned on (*TRACE ON*) and, of course, form one's own opinion.

In what follows, the last option is pursued. The expectations of what the classes can do and what responsibilities can be assigned to them will guide the reflection process. Accomplishing this task with just the own mind fosters a deep understanding of how a knowledge base is processed. Alternatively, a good book, a web reference or even the source code would be a valuable tool to perhaps accomplish this task with less effort and in less time.

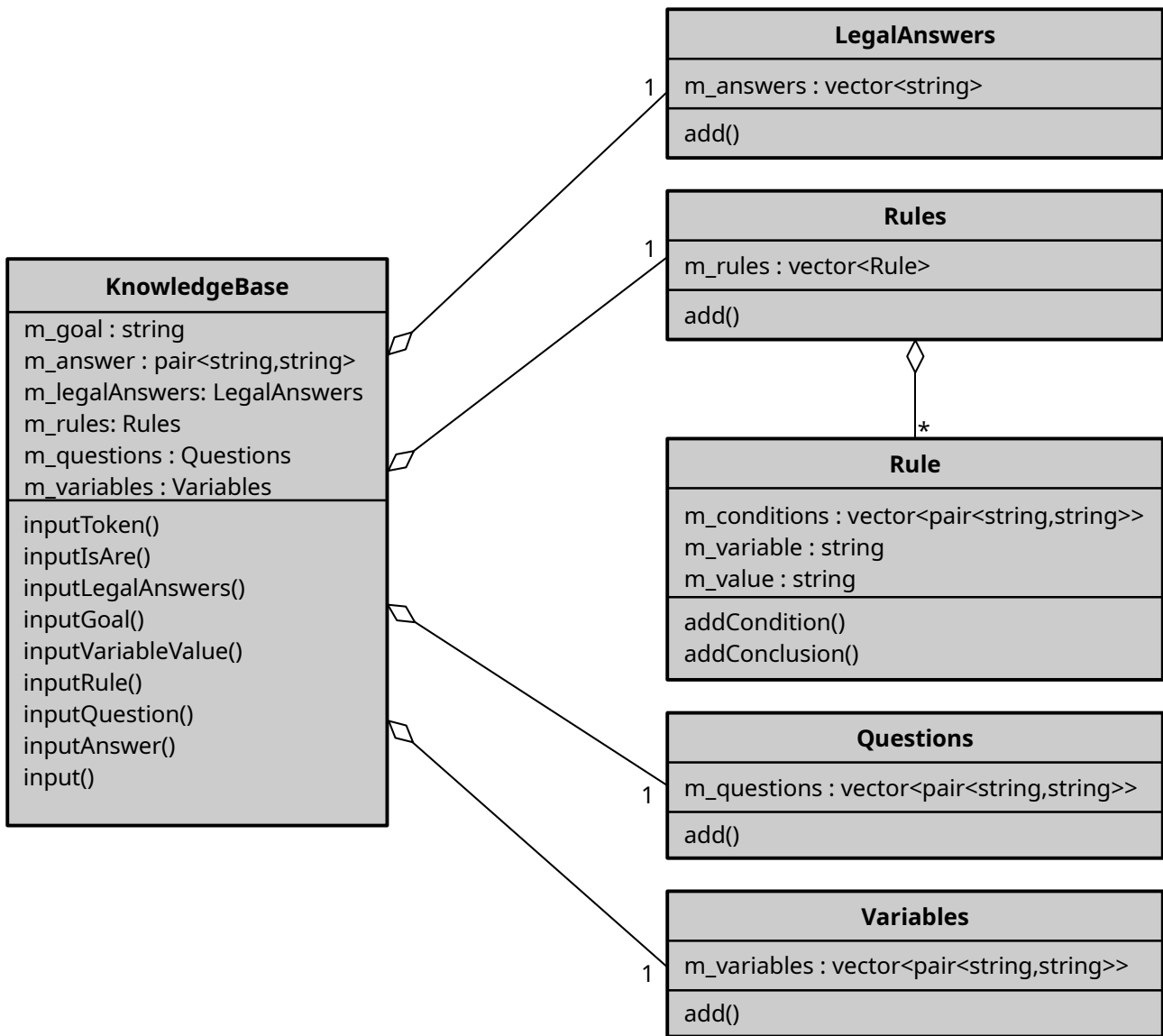


Figure 60: Class diagram with integrated data and loading perspective

When a knowledge base is fully loaded and running, it attempts to assign a value to the *goal* variable. The attempt to assign a value to a variable is called *prove*. Consequently, the `prove()` method is added to the `KnowledgeBase`. Listing 150 shows its prototype.

```
1 bool KnowledgeBase::prove(const string& variable);
```

Listing 150: Declaration of `KnowledgeBase::prove()`

This method returns `true` if it can assign a value to the variable, and `false` otherwise. What can be a useful implementation of this method? Well, `KnowledgeBase::prove()` can ask `Rules` to prove the variable. Therefore, `Rules` must also have a `prove()` method. Since `Rules` does not have access to the knowledge base's variable store, i.e. `KnowledgeBase::m_variables`, the knowledge base must pass itself as an additional parameter. Listing 151 shows the prototype of this method.


```
1 bool Rules::prove(KnowledgeBase& kb, const string& variable);
```

Listing 151: Declaration of Rules::prove()

This method returns true if a value has been assigned to the variable and false if not.

To illustrate the sequence of actions, a new *UML* diagram is introduced, namely the *sequence diagram*. The diagram in Figure 61 shows a knowledge base and its `m_rules` attribute. Since the name of the knowledge base is not important, its name in the upper left object box is `:KnowledgeBase`. This denotes an anonymous object. If this object had a name, it would appear before the colon. The second object box stands for `m_rules:Rules`. This object has a name, namely `m_rules`. The dashed lines pointing down from the object boxes are so-called *lifelines*. Time runs from top to bottom. In Figure 61, both objects exist for the entire time represented by the diagram. This may be different in other scenarios. The left arrow, which starts from a small filled circle, represents a so-called *found message*, i.e. the sender of the message is not relevant. The name of the message is in its label, here `prove()`. When the message hits the `:KnowledgeBase` lifeline, it starts an activity. This activity is symbolized by a so-called *activity bar*. The activity bar extends as long as the processing of the method continues. As a result of receiving this message, `:KnowledgeBase` itself sends a message to `m_rules`, namely `prove()`. The processing of this message is marked by another activity bar that starts at the point where the arrow begins and extends to the end of the activity.

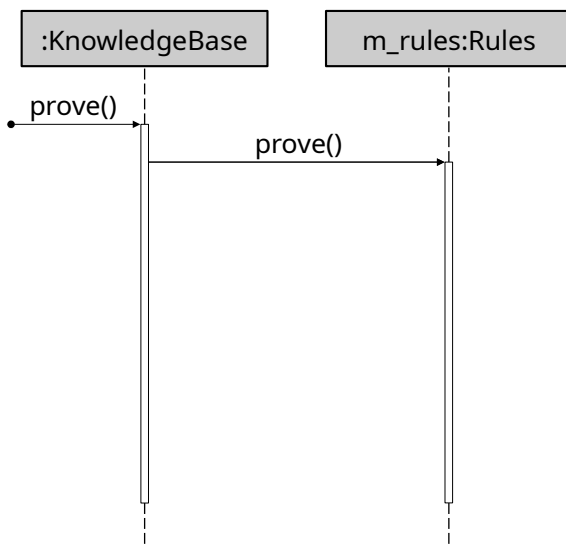


Figure 61: Sequence diagram for starting `prove()`

This diagram is not yet complete. More participating objects will be added and more symbols will be introduced next.

An exemplar of Rules maintains exemplars of Rule. Thus, `Rules::prove()` iterates over its rules and asks each rule if it can prove the variable. If a rule can

successfully assign, i.e. *bind* a value to the variable, `Rules::prove()` immediately returns true. If no rule can assign a value to the variable, the method returns false. Listing 152 shows the pseudo code for this method.

```

1 Rules::prove
2   input parameters are knowledgebase, variable
3   return bool
4 for (rule in m_rules)
5   if rule.prove(knowledgebase,variable)
6     return true
7 return false;

```

Listing 152: Pseudo code of `Rules::prove()`

Now the attempt to bind a value to a variable is passed from `Rules` to a single `Rule`, as Figure 62 illustrates.

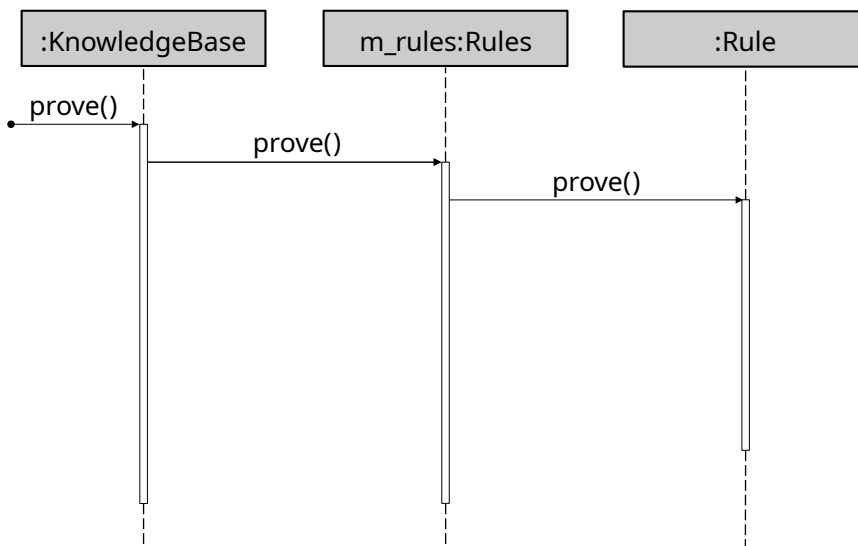


Figure 62: Sequence diagram for further processing of `prove()`

Next, `Rule::prove()` is examined in more detail. Listing 153 shows the prototype of this method. Listing 154 describes its implementation using pseudo code.

```

1 bool Rule::prove(KnowledgeBase& kb, const string& variable);

```

Listing 153: Declaration of `Rule::prove()`

```

1 // knowledge base and variable are function parameters
2 if variable is_not m_variable
3   return false
4 for (condition in m_conditions)
5   value = knowledgebase.getValue(condition.first)
6   if value is empty
7     if not knowledgebase.prove(condition.first) and
8       not knowledgebase.askValue(condition.first)
9     return false
10  value = knowledgebase.getValue(variable)
11  if value is_not condition.second
12    return false
13 knowledgebase.setVariable(m_variable,m_value)
14 return true

```

Listing 154: Pseudo code of Rule::prove()

This method is complex. First it checks if the variable of its conclusion matches `m_variable`. If not, it returns `false` (line 3), because it definitely cannot assign a value to `variable`.

Then the elementary conditions of the rule managed by `Rule::m_conditions` (line 4) are traversed. It should be noted that an elementary condition checks for equality, while all elementary conditions are associated with logical And. All elementary conditions together are called a (complex) condition. However, a (complex) condition can also consist of only one elementary condition. The first step in the loop is to query the knowledge base for the current value of the variable of the current elementary condition, i.e. `condition.first`, by calling `knowledgebase.getValue(condition.first)` (line 5) and assigning the result to `value`. If `value` is the empty string (line 6), the variable (`condition.first`) has not been set yet. In this case, the knowledge base is asked to prove this variable (line 7). If the knowledge base cannot successfully prove the variable, the knowledge base is asked if a question can bind a value to the variable (line 8). The order of calling `KnowledgeBase::prove()` and `KnowledgeBase::askValue()` is chosen according to requirement 3.3.75. Please note how the short-circuit evaluation of Boolean expressions is used to implement this behavior (Section `bool Type`). If asking for a value also fails, the method returns `false` (line 9), because it cannot assign a value to the variable (`condition.first`).

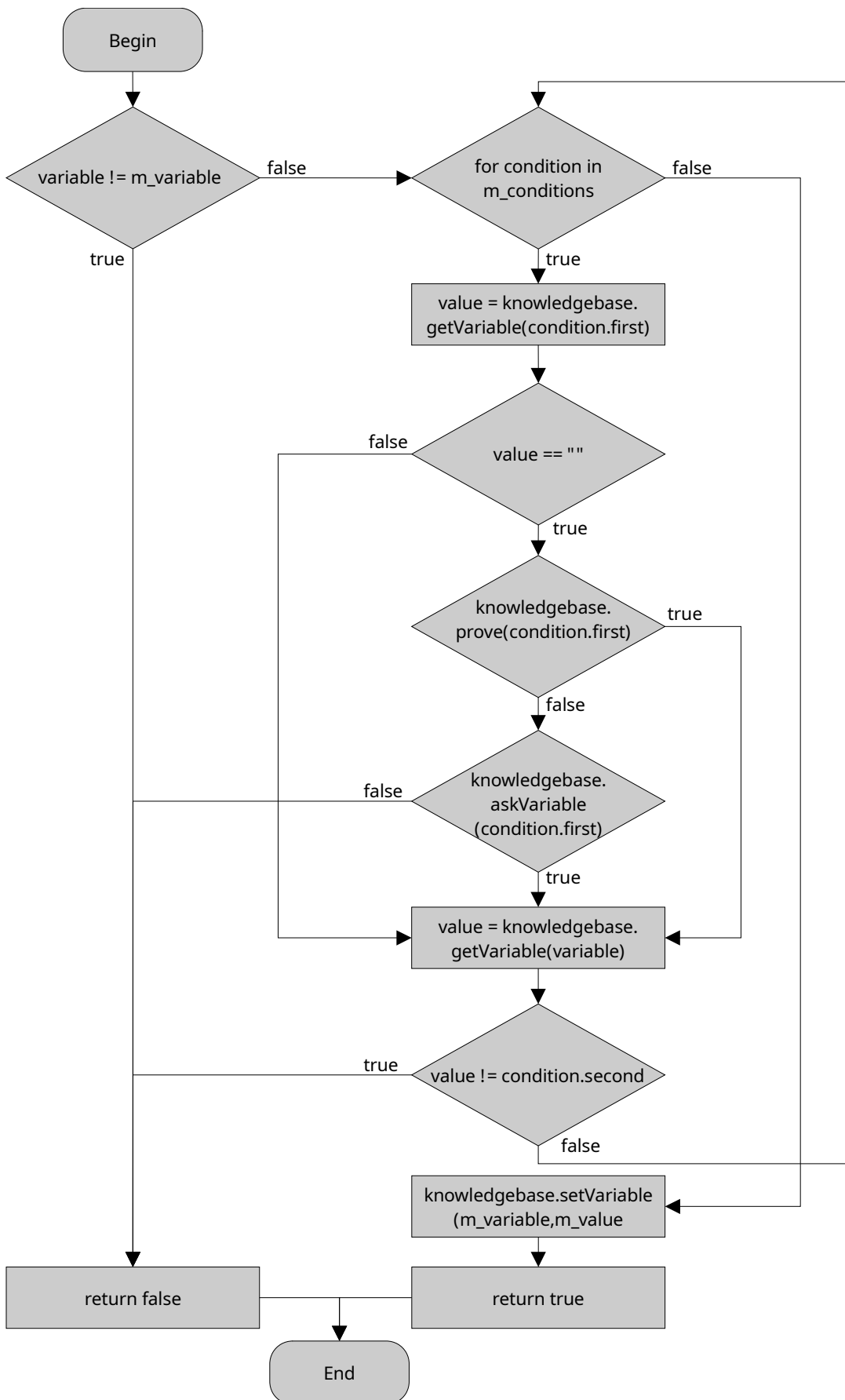
If the knowledge base has assigned a value to the variable (either by proof or by query), the knowledge base must be asked again for the value of the variable (line 10), since it now has a value other than the empty string.

If the determined value differs from the value of the elementary condition (`condition.second`), the method returns `false` (line 12), because this elementary condition is not fulfilled, and therefore the (complex) condition cannot be fulfilled.

Finally, the last elementary condition was evaluated. If the method did not return `false` before, all condition parts are fulfilled. This means that the rule can *fire*, that is, it can assign, i.e. bind the value of its conclusion (`m_value`) to the variable of its conclusion (`m_variable`). This is done by asking the knowledge base to set the variable to the appropriate value (line 13). Ultimately, the method returns `true`.

The flowchart in Figure 63 illustrates the algorithm described for `Rule::prove()`.

Figure 63: Flowchart for the Pseudo code of Rule::prove()



The sequence diagram in Figure 64 gives a *simplified* overview of an example interaction between a KnowledgeBase exemplar, a Rules exemplar, and two exemplars of Rule. After the message `prove()` reaches the left `:Rule`, it is assumed that the left `:Rule` asks the `:KnowledgeBase` to prove a variable, which is illustrated by the arrow leaving the diagram to the right and re-entering from the left. Since this arrow entering from the left does not have a filled circle, it is not a found message. As a consequence of receiving this message, `:KnowledgeBase` asks `m_rules` again to prove the corresponding variable. `m_rules` asks another `:Rule` (the right `:Rule`) to prove the corresponding variable. The right `:Rule` returns to `m_rules` with a dashed line, `m_rules` returns to `:KnowledgeBase` with a dashed line, and `:KnowledgeBase` returns with a dashed line that leaves the diagram to the left but re-enters from the right. After that, there are more dashed lines indicating the returns. This is the end of the activity bars. It should be pointed out that because of indirect recursion, a second activity bar appears for `m_rules`, which is superimposed on top of the first as a slightly thicker rectangle to make it easier to recognize.

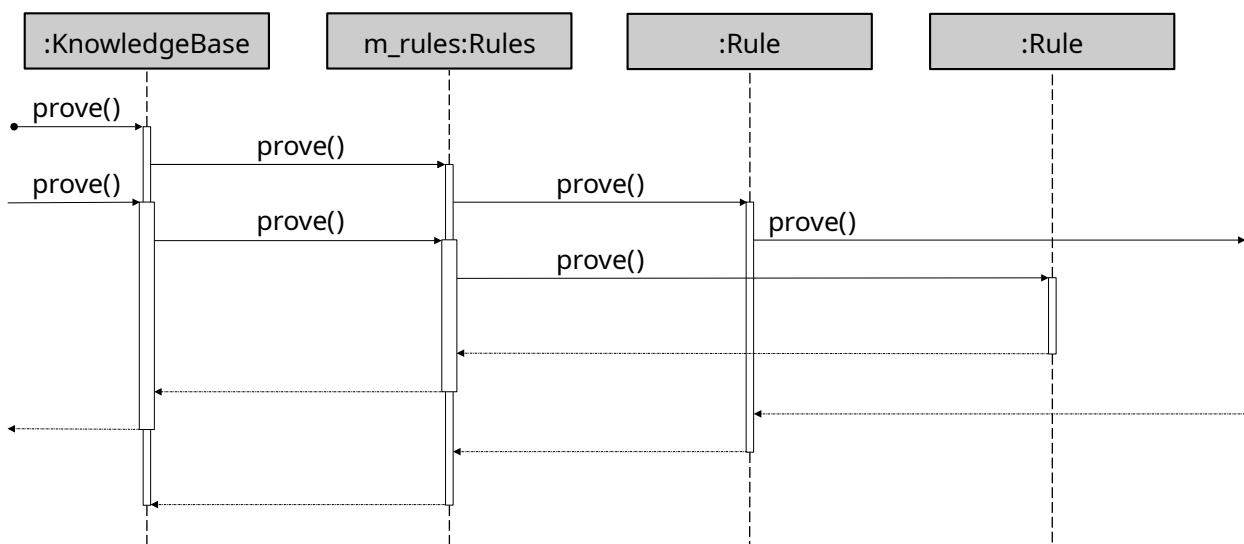


Figure 64: Sequence diagram for the further processing of `prove()`

`Rule::prove()` calls a method that has not yet been introduced, namely `KnowledgeBase::askValue()`. Listing 155 shows the prototype of this method.

```

1 bool askValue(const string& variable);

```

Listing 155: Declaration of `KnowledgeBase::askValue()`

The knowledge base delegates this task to its member `m_questions`, which manages all questions. Since an exemplar of `Questions` has no access to the variable store, the knowledge base must pass itself as a parameter. If no question is found that

assigns a value to variable, the method returns false, otherwise true. Since this method is very simple, no pseudo code for it is shown.

Listing 156 shows the prototype of the `Questions::ask()` method, while Listing 157 shows the pseudo code for its implementation.

```
1 bool Questions::ask(KnowledgeBase& kb, const string& variable);
```

Listing 156: Declaration of Questions::ask()

```
1 // knowledgebase and variable are function parameters
2 string answer
3 for (question in m_questions)
4     if question.first is variable
5         do
6             {
7                 print question.second
8                 input answer
9                 if knowledgeBase.isLegalAnswer(answer)
10                    knowledgebase.setVariable(question.first,answer)
11                    return true
12                else
13                    print Illegal answer
14                    print Legal answers are
15                    knowledgebase.outputLegalAnswers()
16            } until knowledgebase.isLegalAnswer(answer)
17 return false
```

Listing 157: Pseudo code of Questions::ask()

This method iterates through the questions (line 3), looking for a question for variable (line 4). When it finds a question, it outputs its subject, i.e. `question.second` (line 7), and asks for an answer, making sure that the answer is valid (lines 9ff.). Then the value (`answer`) of the variable (`question.first`) is set by calling `knowledgebase.setVariable(question.first,answer)`, and the method returns true (line 11). If no question is found, the method returns false (line 17).

Two methods appear here for the first time, namely `KnowledgeBase::isLegalAnswer()` and `KnowledgeBase::outputLegalAnswers()`. Both methods have simple implementations. The first method simply returns `m_legalAnswers.isLegal(answer)` and the second calls `m_legalAnswers.output()`. Therefore, no pseudo code is specified for them.

Overall, the processing has now been outlined in sufficient detail. Since `KnowledgeBase` indirectly asks a `Rule` to assign a value to a variable, and the `Rule` in turn can ask a `KnowledgeBase` to assign a value to a variable, the algorithm is a case of indirect recursion (see [Recursion](#) Section). In this way, the various function calls build up the function call stack. Whenever a variable is successfully assigned a value, the corresponding function returns and the top entry of the function call stack is removed. Thus, the stack mentioned in the requirements is indirectly realized by the function call stack. It has already been mentioned that recursion should be avoided in C++, since it is not primarily designed as a recursive programming language. In this case, the recursive implementation will serve as a

reference point for possible future iterative implementations. When both implementations of the inference process, recursive and iterative, are available, it might be instructive to compare them in terms of performance.

All methods involved in processing must be assigned to the appropriate classes in the class diagram. The class diagram in Figure 65 shows all members involved in data, loading and processing.

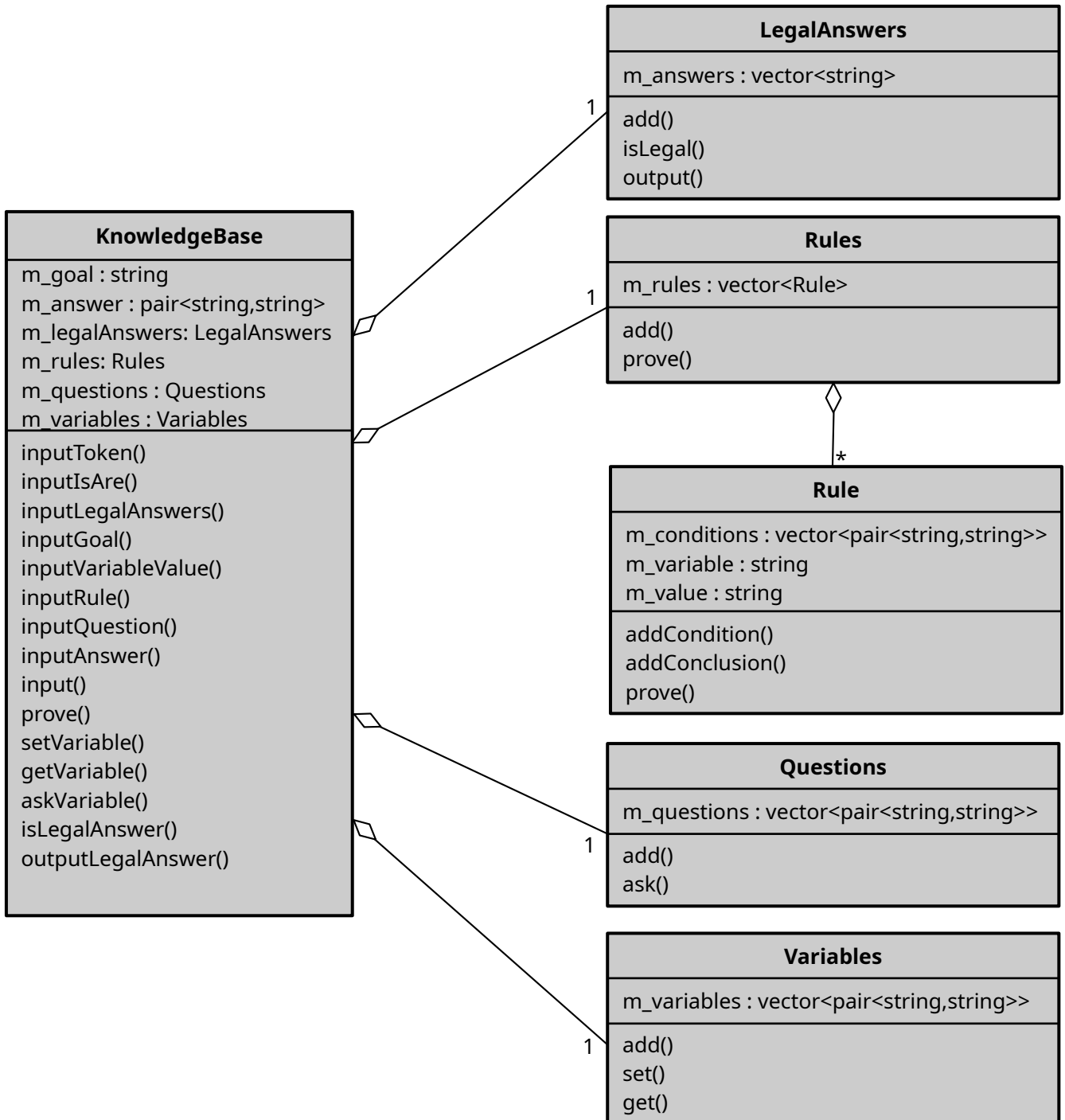


Figure 65: Class diagram with integration of all perspectives

6.5. Implementation

The following shows the implementation of the different entities in terms of their dependencies. An entity with no or less dependencies is shown before an entity with more dependencies.

If applicable, each implementation part is discussed in terms of data, loading and processing. Emerging requirements and important improvements are presented as they occur. This is the maximum amount of order that can be achieved without giving a false impression of the development of a similar program in practice. In fact, the development of a program would appear rather chaotic, depending on the experience of the software developers and their personal knowledge of the problem domain of the program. However, it is precisely iterative and incremental development that is at the core of *agile software development* methods, which are the de facto standard for software development today.

Since the source code contains more than 800 lines, the program is presented only in excerpts. Otherwise, reading and understanding it would become too difficult. The complete program is contained in *ec/EC.cpp*. It is recommended to load it into an editor and browse through it while reading the following text.

6.5.1. Include Files

The Listing 158 shows the initial comment, the includes, and the import of the namespace std.

```
1 // EC.cpp by Ulrich Eisenecker, January 25, 2024
2
3 #include <iostream>
4 #include <fstream>
5 #include <string>
6 #include <algorithm>
7 #include <vector>
8 #include <utility> // because of pair<>
9 #include <cctype> // because of toupper()
```

Listing 158: EC.cpp – included files

The first line of the program is a comment stating the name of the program, its author and the date of completion. The program is called *EC*, which is an acronym for *ESIE™ Cover*, since it can be considered a cover version of the original *ESIE™*.

The header file `<iostream>` is required for general input and output. The user interface of *EC* is implemented with common console input and output. This differs from *ESIE™*, which uses text output and input in graphics mode. However, the content and sequence of *EC*'s input and output are exactly the same as *ESIE™*.

The `<fstream>` header file must be included because loading a knowledge base involves processing files.

All strings are of type `std::string`. Therefore `<string>` is included.

As mentioned several times, the implementation of *EC* makes intensive use of the *Standard Template Library* (abbreviated *STL*), which is part of the C++ standard library. Algorithms of the STL are contained in `<algorithm>`, while the header file `<vector>` provides the container template `std::vector<>`, which is a kind of Swiss Army knife of all container types.

The struct template `std::pair<>` is explained in the Data Section. To use it, `<utility>` is included.

Since the `toupper()` function is used in `KnowledgeBase::inputToken()`, `<cctype>` is included.

Finally, using `namespace std;` imports all identifiers of the namespace `std`, which contains all entities of the previously included header files. This global namespace import is acceptable because the implementation of *EC* is contained in a single large file.

6.5.2. Logger Class

When implementing the processing of a knowledge base, replication of the *ESIE*[™] tracing messages proved to be a serious challenge. Interestingly, the processing algorithm implemented in the various methods named `prove()` is relatively insensitive to change. In addition, it was not immediately clear at which point in the code the trace messages should be issued. To better understand the actual processing, debugging the program and examining the function call stack was not very helpful because it took a long time to display the processing steps of interest and the amount of information presented was overwhelming. Therefore, another way to understand the processing was needed. The necessary functionality is implemented in the `Logger` class, shown in the Listing 159. It introduces some new features.

```
1 constexpr bool loggingActive { false };
2
3 // Logger is for logging only
4 class Logger
5 {
6     public:
7         Logger(const string& name, const string& arguments = ""):
8             m_name { name }, m_arguments { arguments }
9         {
10             if constexpr (loggingActive)
11             {
12                 string indentation(++m_activeFunctions, '>');
13                 cout << indentation << ' '
14                     << m_name << ' ';
```

```

15         << m_arguments << endl;
16     }
17 }
18 ~Logger()
19 {
20     if constexpr (loggingActive)
21     {
22         string indentation(m_activeFunctions--,'<');
23         cout << indentation << ' '
24              << m_name << ' '
25              << m_arguments << endl;
26     }
27 }
28 private:
29     static inline size_t m_activeFunctions { 0 };
30     const string m_name { },
31                m_arguments { };
32 };

```

Listing 159: EC.cpp – Logger class

First there is the global constant `loggingActive`, which is of type `bool` and is initialized with `false`. `false` means that no logging output is generated. If `loggingActive` is initialized to `true`, logging output is generated as intended.

Next is the declaration with `constexpr`. `constexpr` means that the declared object is fully evaluated at compile time, i.e. the compiler computes and knows its value. Here this is obvious because `loggingActive` is immediately initialized with a `bool` constant. Any object declared with `constexpr` is also `const`, i.e. it cannot be changed after its declaration. Conversely, this is not necessarily the case, because a `const` object can also be created or become a `const` object at runtime. In the following it becomes clear why this is important.

The purpose of the `Logger` class is to provide information about entering and exiting a function, outputting the name of the function and the current values of its parameters. Depending on the nesting level of the function calls, the output is to be indented accordingly. For the indentation when entering a function the `>` character is to be used, for the indentation when leaving a function the `<` character.

The `Logger` class has three private data members, namely a so-called *static data member* `Logger::m_activeFunctions` of type `size_t`, which contains the current number of exemplars of `Logger` (it corresponds to the number of active functions on the call stack), and two ordinary `const` data members of type `std::string`, namely `Logger::m_name` and `Logger::m_arguments`.

Exemplars of `Logger` can be created multiple times in different places of a program. Since the intended indentation of logging messages depends on the actual number of exemplars of `Logger`, the number of exemplars must be tracked. This number is valid only per class, not per exemplar. Declaring a data member of a class as `static` has the effect that the data element exists only once per class. One can think of it as being bound to the class itself, but not to an exemplar. Thus, `static inline size_t m_activeFunctions { 0 };` declares `Logger::m_activeFunctions`

as a data member that exists in the `Logger` class, but not in its exemplars. Of course, a static class variable can be accessed from anywhere in the class, but it is always the same class variable. The additional use of `inline` allows the static data element to be initialized in the class at the point where it was declared, here with `{ 0 }`.

`Logger` has only two public methods, namely a constructor and a destructor. The constructor takes two parameters, `name` and `arguments`, which are used to initialize the member variables `Logger::m_name` and `Logger::m_arguments`. There is also a default value for the parameter `arguments`, namely the empty string `""`. So to log a function without parameters, an exemplar of `Logger` can be created by specifying only the name of the function without arguments.

The first statement in the constructor is `if constexpr (loggingActive)`. This is a variant of `if` that is evaluated at compile time. Therefore, the following `if` statement is preserved if `loggingActive` is `true`. The `if` condition itself is eliminated in this case. That is, the result of `if constexpr (true) someStatement;` is simply `someStatement;`. If `loggingActive` is `false`, neither the condition nor the dependent statement will be present in the compiled code, i.e. `if constexpr (false) someStatement;` becomes the empty statement `;`. The same technique is used in the destructor. Thus, if `loggingActive` is `false`, no code is generated by the compiler other than initializing and destroying the data members of a logger exemplar. If the compiler performs very aggressive optimizations, it is even conceivable that the generation of logger exemplars is omitted altogether because they are not used in the program at all.

If `loggingActive` is `true`, the `std::string` `indentation` is generated and initialized with `Logger_m_activeFunctions` repeats of `'>'` by calling the so-called *fill constructor* of `std::string`. Before this, `Logger::m_activeFunctions` is incremented by one. This has the effect that indentation level 1 is output with exactly one trailing `'>'`.

The next statement outputs the contents of `indentation` followed by a space, `Logger::m_name` followed by a space, and `Logger::m_arguments` followed by a new line.

The implementation of the destructor is parallel to that of the constructor. Since the corresponding function is exited, `Logger::_m_activeFunctions` is decremented by one after initializing the indentation, and `'<'` is used as a terminator to indicate that the function is left.

Listing 160 illustrates the purpose and use of `Logger`. The `main()` and `factorial()` functions declare exemplars of `Logger`. When the program is compiled and executed, it asks for a number to calculate the factorial. It then reports the calls to `main()` and the recursive `factorial()` function. The definition of the `Logger` class is exactly the same as in Listing 159, so it is not shown again.

```

1 // LoggerDemo.cpp by Ulrich Eisenecker, January 5, 2024
2
3 #include <iostream>
4 #include <string>
5 #include <cstdint> // Because of intmax_t
6
7 using namespace std;
8
9 constexpr bool loggingActive { false };
10
11 // Logger is for logging only
12 class Logger
13 {
14     public:
15         Logger(const string& name,const string& arguments = ""s):
16             m_name { name },m_arguments { arguments }
17         {
18             if constexpr (loggingActive)
19             {
20                 string indentation(++m_activeFunctions, '>');
21                 cout << indentation << ' '
22                     << m_name << ' '
23                     << m_arguments << endl;
24             }
25         }
26         ~Logger()
27         {
28             if constexpr (loggingActive)
29             {
30                 string indentation(m_activeFunctions--, '<');
31                 cout << indentation << ' '
32                     << m_name << ' '
33                     << m_arguments << endl;
34             }
35         }
36     private:
37         static inline size_t m_activeFunctions { 0 };
38         const string m_name { },
39                 m_arguments { };
40 };
41
42 intmax_t factorial(intmax_t n)
43 {
44     Logger log { "factorial",to_string(n) };
45     if (n == 0)
46     {
47         return 1;
48     }
49     else
50     {
51         return n * factorial(n - 1);
52     }
53 }
54
55 int main()
56 {
57     Logger log { "main"s };
58     intmax_t n;
59     cin >> n;
60     cout << factorial(n) << endl;
61 }

```

Listing 160: `ec/LoggerDemo.cpp` (excerpt)

The `to_string()` function converts an integer value to a `std::string` and has already been introduced.

Figure 66 shows an example dialog with logging enabled calculating the factorial of 5. The user input is formatted in **blue font**.

```
> main
5
>> factorial 5
>>> factorial 4
>>>> factorial 3
>>>>> factorial 2
>>>>>> factorial 1
>>>>>>> factorial 0
<<<<<<< factorial 0
<<<<<< factorial 1
<<<<< factorial 2
<<<< factorial 3
<<<< factorial 4
<< factorial 5
<
1
< main
```

Figure 66: Program execution with logging enabled

It should be mentioned that there are several libraries and frameworks for logging in C++. To spare a more comprehensive introduction, an own implementation of a very specific and lightweight Logger class was chosen instead.

6.5.3. toupper() Function

Listing 161 shows the definition of a small helper function that converts lowercase ASCII letters contained in the `std::string` passed as an argument to the corresponding uppercase letters and returns the result. The implementation of this function does not recognize other character encodings. Therefore, its scope is limited to the `ec/EC.cpp` file. It should be emphasized that `toupper(string)` and `toupper(char)` are two different functions that have the same name due to function overloading.

```
1 // Converts lower cases to upper cases in ASCII character set
2 [[nodiscard]] string toupper(string s)
3 {
4     for (char& c : s)
5         c = toupper(c);
6     return s;
7 }
```

Listing 161: EC.cpp – toupper() function

6.5.4. LegalAnswers Class

Listing 162 shows the LegalAnswers class. Its first data member is an instantiation of the container template `std::vector<>` for type `std::string` named `m_answers`. Its second data member, `m_active`, is of type `bool` and is initialized to `false`. Why this is important will be explained later.

```
1 class LegalAnswers
2 {
3     public:
4         [[nodiscard]] size_t size() const
5         {
6             return m_answers.size();
7         }
8         bool add(const string& answer)
9         {
10            m_answers.push_back(answer);
11            return true;
12        }
13        void setActive()
14        {
15            m_active = true;
16        }
17        [[nodiscard]] bool isActive() const
18        {
19            return m_active;
20        }
21        [[nodiscard]] bool isLegal(const string& answer) const
22        {
23            if (!isActive())
24            {
25                return true;
26            }
27            else
28            {
29                return find(m_answers.begin(),m_answers.end(),answer) !=
30                    m_answers.end();
31            }
32        }
33        void output() const
34        {
35            if (m_answers.empty())
36            {
37                return;
38            }
39            for (const auto& element : m_answers)
40            {
41                cout << element << endl;
42            }
43        }
44        void output(ostream& os) const
45        {
46            if (m_answers.empty())
47            {
48                return;
49            }
50            os << "legalanswers are "s;
51            for (const auto& element : m_answers)
52            {
53                os << element << ' ';
54            }
55            os << "*\n"s << endl;
56        }
57    private:
```

```

58     vector<string> m_answers;
59     bool m_active { false };
60 };

```

Listing 162: *EC.cpp* – *LegalAnswers* class

The `LegalAnswers::size()` member function simply calls `vector<string>::size()` for its `LegalAnswers::m_answers` data member, i.e. it *forwards* this message. Alternatively, the free function template `size<>()` could be called, which in turn also calls `vector<>::size()`. The `size<>()` function template is overloaded with the same semantics for other container templates as well. The `LegalAnswers::add()` method adds the argument `answer` of type `std::string` to `LegalAnswers::m_answers` by calling `m_answers.push_back(answer);`. The member function `vector<>::push_back()` inserts its argument at the end of the `vector<>`.

There are two important aspects to note here. First, `answer` is not itself inserted into the `vector<>`. Rather, it is copied and this copy becomes the last element of the `vector<>`. Second, when a `vector<>` is created it is initialized with a default capacity of elements.

When the insertion of a new element exceeds the actual capacity of a `vector<>`, the `vector<>` automatically grows so that the new element can be safely inserted. Therefore, `LegalAnswers::add()` can never fail and always returns `true`. For that reason, one could argue that `LegalAnswers::add()` should return `void`. To support ignoring the return value of the function, it is not prefixed with `[[nodiscard]]`. This way it can be called as if it were returning `void`. However, the return type `bool` was chosen intentionally. In a future revised version of *ec/EC.cpp*, it would be advisable to use a different container template that prevents duplicate entries, e.g. `std::set<>`. In a `set<>`, elements occur at most once. Unlike a mathematical set, a `set<>` is internally ordered, which allows fast searching of elements. Before adding a new answer, it should be checked if the value of `answer` has already been set. If this is the case, the member function should return `false` to indicate this problem. This would allow a duplicate answer to be treated as a knowledge base error, and at least an appropriate warning should be issued. *ec/EC.cpp* is only implemented to be fully *ESIE*[™] compliant, which also does not check for this error.

When loading a knowledge base, `LegalAnswers::setActive()` must be called as soon as a *LEGALANSWERS* rule starts. This method sets `LegalAnswers::m_active` to `true`, indicating that a *LEGALANSWERS* rule has occurred. `LegalAnswers::isActive()` returns the current value of `LegalAnswers::m_active`. This can be used to query whether a *LEGALANSWERS* rule has already been found. This option is used in `LegalAnswers::isLegal()`.

If no *LEGALANSWERS* rule is present, `LegalAnswers::isActive()` returns `false`. In this case, the `LegalAnswers::isLegal()` method returns `true` because any answer is acceptable. Otherwise, the `LegalAnswers::isLegal()` method returns the result of

the answer search in `LegalAnswers::m_answers`. This enables the inherently incorrect behavior of *ESIE*[™] to start an infinite loop when no values are specified in the *LEGALANSWERS* rule (see requirement with ID 1.5.7).

The algorithm `std::find<>()` is a function template that takes three arguments. The first argument is an iterator pointing to the first element of the search range, its second argument is an iterator pointing to the position immediately after the last element of the search range, and the third parameter is the value being searched for. An iterator can be thought of as a pointer to an element of a container template. As such, it can be dereferenced to get the corresponding element. But unlike raw pointers in C++, the programmer does not have to worry about obtaining or disposing of memory. `m_answers.begin()` returns an iterator that points to the first element of `LegalAnswers::m_answers`. The public member function `container<>.begin()` is provided by most of the STL container templates. `m_answers.end()` returns an iterator pointing to the position immediately after the last element of `LegalAnswers::m_answers`. It cannot point to the last element of a `vector<>`, as this would prevent empty vectors. Therefore, this particular position immediately after the last element is required, as it also applies to an empty container. `find<>` returns an iterator pointing to the element if it is found, and to `vector<>.end()` otherwise. For this reason, it is checked whether `find(m_answers.begin(),m_answers.end(),answer)` is equal to `m_answers.end()`. The result of this comparison is immediately returned as the result of the member function.

The `LegalAnswers::output()` member function has no arguments and is called while the knowledge base is being processed, i.e. during a consultation. When the user enters an invalid answer, all legal answers are output, each on a separate line. This is exactly what this method does.

The `LegalAnswers::output()` member function, which takes a reference to `ostream` as an argument, does not implement a requirement. It was introduced and implemented in the context of testing the loading of a knowledge base. How can one be sure that a knowledge base is loaded correctly? One might be able to check this with a debugger by inspecting the contents of the various relevant variables, but only for small knowledge bases and in a cumbersome way. But there is a solution!

After loading a knowledge base, its contents can be output to a text file in a valid knowledge base format. Unfortunately, the original knowledge base and the exported knowledge base can differ significantly. This is because the exact order of the different rules, their formatting, and the use of lower and upper case letters are lost during loading.

However, the idea is to reload the exported knowledge base and export it a second time. After that, the first and the second export must be identical. If this is not the case, either the input or the output may be incorrect. If the code that produces the

output has been carefully checked, the likelihood that the errors are in the code that implements the input increases. Input and output of a knowledge base are not symmetrical in terms of format and implementation. While it is relatively difficult to implement all the methods required for input, it is relatively easy to implement the necessary output methods. For this reason, corresponding output methods were added to all objects relevant to the knowledge base. In fact, some errors could be found by reproducing the loaded knowledge bases through their output, but this will not be discussed further here.

`LegalAnswers::output(ostream& os)` first checks if the data member `m_answers` is empty. If yes, it returns immediately. Otherwise, a range-based for loop is used to output each answer to `os`, followed by a single space. Finally, a terminating asterisk, `'*'`, is output to `os`.

Overall, `LegalAnswers` is a lightweight class that adds little functionality to the `std::vector<>` container template. Using a `vector<>` instead of `LegalAnswers` and implementing the additional functionality in place would have been a viable option. Doing so would have been one less class to write, understand, and maintain. Nevertheless, the decision was made to implement it as a separate class, creating a domain-specific abstraction of the solution space and increasing the intentionality of the code.

6.5.5. Variables Class

Listing 163 shows the implementation of the `Variables` class. It manages all variables that occur in the rules of a knowledge base.

```
1 class Variables
2 {
3     public:
4         [[nodiscard]] size_t size() const
5         {
6             return m_variables.size();
7         }
8         bool add(const string& variable)
9         {
10            for (const auto& element : m_variables)
11            {
12                if (element.first == variable)
13                {
14                    return false;
15                }
16            }
17            m_variables.push_back(make_pair(variable, ""));
18            return true;
19        }
20        bool set(const string& variable, const string& value)
21        {
22            for (auto& element : m_variables)
23            {
24                if (element.first == variable)
25                {
26                    element.second = value;
```

```

27         return true;
28     }
29 }
30 return false;
31 }
32 [[nodiscard]] string get(const string& variable) const
33 {
34     for (const auto& element : m_variables)
35     {
36         if (element.first == variable)
37         {
38             return element.second;
39         }
40     }
41     return "";
42 }
43 void reset()
44 {
45     for (auto& element : m_variables)
46     {
47         element.second = "";
48     }
49 }
50 void output(ostream& os) const
51 {
52     for (const auto& element : m_variables)
53     {
54         os << element.first << " is "
55         << element.second << endl;
56     }
57 }
58 private:
59     vector<pair<string,string>> m_variables;
60 };

```

Listing 163: *EC.cpp* – Variables class

A variable is a simple concept that combines a *name* and a *value*, both of which are strings. This does not justify its implementation as a domain-specific concept. Fortunately, the C++ standard library provides the struct template `std::pair<>` that can be used to represent a variable, i.e. `pair<string,string>`. Assuming `element` is a variable of type `pair<typeA,typeB>`, `element.first` is the first pair component of typeA, and `element.second` is the second pair component of typeB. There are no isolated variables in the program, they are all managed in a particular store. The Variables class serves exactly this purpose. It has only one member, namely `m_variables` of type `vector<pair<string,string>>`. The `vector<>` container template was chosen because the requirements and various tests showed that a similar data structure is likely to be ubiquitous in *ESIE*[™]. In particular, the order in which variables occur is preserved by storing them in a `vector<>`.

Having completed the implementation of the cover version of *ESIE*[™], it appears that this feature is probably not relevant. In a revised version, a more appropriate container template could be used, for example `std::map<>`. A `map<>` associates *keys* and *values*. Each key-value pair is stored internally as a `std::pair<>`. The name of a variable would act as the key, and the value associated with the key would act as the

value of the variable. A `std::map<>` automatically ensures that there are no duplicates of key-value pairs.

Next, the methods of `Variables` are discussed.

`Variables::size()` simply forwards the call to `vector<>::size()` and returns its result.

`Variables::add()` adds the parameter variable to `Variables::m_variables`. To avoid duplicate variables, a range-based for loop is executed first, which immediately returns `false` if the variable already exists. It should be noted that the for loop iterates over elements of type `pair<string,string>`. Therefore, `element.first` refers to the *name* of a variable, while `element.second` refers to its *value*. If the for loop ends without returning, the variable is added by calling `m_variables.push_back(make_pair(variable, ""s));`

The `std::make_pair<>()` function template is a helper function that returns a `pair<>` that is correctly instantiated with the types of its two arguments, here `pair<string,string>`. Subsequently, the method returns `true`. `Variables::add()` returns a `bool` value, but it has no `[[nodiscard]]` attribute. That is, the caller can check whether a new variable was successfully stored or not, but the return value can also be ignored.

`Variables::set()` sets variable to value, both of which are passed as parameters. Again, a range-based for loop is used to search for variable in `Variables::m_variables`. If it is found, its value is set by executing `element.second = value;`. After that, the loop is exited by returning `true`. If the loop ends and the variable is not found, nothing is set and the method returns `false`. In this way, only variables that exist in the variable store can be assigned a value. Whether this could be done successfully is indicated by the return value of the method. If the caller does not care about the assignment to a non-existing variable, the return value can be ignored because the member function is not prefixed with the `[[nodiscard]]` attribute. The method could also be designed differently. By assigning a value to a non-existent variable, the variable could be implicitly stored along with its value in `Variables::set()`. This would always succeed, so no return value would need to be specified. Such a construction could be useful in a different context of use.

`Variables::get()` retrieves the value of variable passed as a parameter and returns it as a value of type `std::string`. Again, a range-based for loop searches for variable. If it is found, its value is returned by `return element.second;`. If the variable has not been assigned a value before, `element.second` is an empty string. If the variable cannot be found, the method must still return a string. Therefore, it returns an empty `std::string`, namely `""s`. This return value means that *the variable passed to Variables::get() has not been assigned yet*. Actually, this is wrong, because the variable does not exist. This can be tolerated only because

variables and values are always of type `std::string`, and the interpretation of the return value ""s as either *variable has no value* or *variable does not exist* does not matter in the context of the program. Other application contexts may require more precise solutions.

Since the return value of this method must not be ignored, the `[[nodiscard]]` attribute is prepended to the method.

`Variables::reset()` is a member function added later. When a consultation is finished, the user can enter *GO* again to start a new consultation. In this case all variable values must have been reset to the empty string. Otherwise, the consultation would not be executed correctly. Therefore, `Variables::reset()` is called immediately after a consultation is finished.

`Variables::output()` outputs all variables and their associated values to `os` passed as `ostream` reference. This way it is possible to get a list of all variables declared in the knowledge base rules with their current values. This is useful for monitoring the loading and processing of the knowledge base. This method is not relevant for exporting a knowledge base to a text file, because variables are not explicitly declared in a knowledge base!

Like `LegalAnswers`, `Variables` is a very lightweight class. It can be considered a wrapper that adds little functionality to `Variables::m_variables`. It could have been implemented in its entirety directly in the `KnowledgeBase` class. Its advantage is that it introduces an intentional name for an important domain-specific concept. This is different for *variable*. Since there are no isolated variables in the program, it would not be justifiable to introduce a `Variable` class for it just to give it a domain-specific flavor.

6.5.6. Pending Declarations

Sometimes it is necessary for objects of different types to use each other, e.g. by passing them as parameters to member functions. In C++, however, only types that are already known to the compiler can be used in another declaration.

In `ec/EC.cpp` this situation occurs three times: Each of the classes `Questions`, `Rule` and `Rules` uses the class `KnowledgeBase` in one of its methods. Therefore, a so-called *pending declaration* of `KnowledgeBase` is required before the `Questions` class is defined, namely `class KnowledgeBase;`

To better understand this problem and its solution, Listing 164 illustrates this situation with the two classes `Ping` and `Pong` which have methods that take exemplars of each other as parameters.

```
1 // PingPongImpossible.cpp by Ulrich Eisenecker, August 31, 2021
2 // Caution: This program does not compile!
```

```

3
4 class Ping
5 {
6     public:
7         void method(Pong& p,int n)
8         {
9             if (n > 0)
10            {
11                p.method(*this,n - 1);
12            }
13        }
14 };
15
16 class Pong
17 {
18     public:
19         void method(Ping& p,int n)
20         {
21             if (n > 0)
22             {
23                 p.method(*this,n - 1);
24             }
25         }
26 };
27
28 int main()
29 {
30     Ping ping;
31     Pong pong;
32     ping.method(pong,3);
33 }

```

Listing 164: *PingPongImpossible.cpp*

When compiling, the compiler reports an error in line 7 because Pong has not yet been defined. The problem can be solved by introducing the name Pong and telling the compiler that it is a class before declaring the Ping class. This is achieved by the *pending declaration*, which can be seen in Listing 165 in line 3.

```

1 // PingPong.cpp by Ulrich Eisenecker, August 31, 2021
2
3 class Pong;
4
5 class Ping
6 {
7     public:
8         void method(Pong& p,int n);
9 };
10
11 class Pong
12 {
13     public:
14         void method(Ping& p,int n)
15         {
16             if (n > 0)
17             {
18                 p.method(*this,n - 1);
19             }
20         }
21 };
22
23 void Ping::method(Pong& p,int n)
24 {
25     if (n > 0)
26     {

```

```

27     p.method(*this,n - 1);
28     }
29 }
30
31
32 int main()
33 {
34     Ping ping;
35     Pong pong;
36     ping.method(pong,3);
37 }

```

Listing 165: *PingPong.cpp*

After that, the program still won't compile if the definition of `Ping::method()` is included in the definition of `Ping`, as it is the case in Listing 164. The reason for this is that the compiler only knows that `Pong` is a class, but it knows nothing about the size of `Pong` and its members, which are accessed in `Ping::method()`. Therefore, only the declaration of `Ping::method()` must be included in the definition of class `Ping`. Its definition must come after the definition of the class `Pong`, which is the case in Listing 165.

Of course, this problem does not occur in the definition of `Pong::method()`, since the definition of `Ping` is already available to the compiler when `Pong::method()` is compiled.

6.5.7. Questions Class

The `Questions` class manages individual questions that ask for the value of a variable. Isolated questions do not occur in the program. Therefore, it is not necessary to define an extra `Question` class. A question consists of a *variable* and a *subject*. The subject is the text of the question to be asked to the user. These two components can be easily mapped to the utility template `std::pair<>`. The only data element of `Questions` is `Questions::m_questions`, which is of type `vector<pair<string,string>>`. Again, the container template `std::map<>` would be more appropriate for managing questions. In this case, the variable name would act as the key and the topic as the value. Besides efficiency, another argument for using `map<>` is that it automatically ensures that there is only one question per variable. On the other hand, `map<>` uses an internal sorting for its elements, while the elements of a `std::vector<>` are sorted externally. Therefore, `vector<>` was chosen to implement this first cover version because it maintains the order of the questions in the knowledge base.

Let `element` be of type `pair<string,string>`. Then `element.first` corresponds to the *name* of the variable and `element.second` to the *subject* of the question. Listing 166 shows the definition of `Questions`. It should be noted that the separate definition of `Questions::ask()` in the source code of *ec/EC.cpp* appears after the

definition of the KnowledgeBase class, because it calls methods of KnowledgeBase that are not yet known when Questions is defined.

```

1 class Questions
2 {
3     public:
4         [[nodiscard]] size_t size() const
5         {
6             return m_questions.size();
7         }
8         bool add(const string& variable, const string& subject)
9         {
10            for (const auto& element : m_questions)
11            {
12                if (element.first == variable)
13                {
14                    return false;
15                }
16            }
17            m_questions.push_back(make_pair(variable, subject));
18            return true;
19        }
20        void output(ostream& os) const
21        {
22            for (const auto& element : m_questions)
23            {
24                os << "question "s << element.first << " is\n\"s
25                << element.second << "\n\"s << endl;
26            }
27        }
28        bool ask(KnowledgeBase& kb, const string& variable);
29    private:
30        vector<pair<string, string>> m_questions;
31 };
32 // ...
33 bool Questions::ask(KnowledgeBase& kb, const string& variable)
34 {
35     Logger log("Questions::ask"s, variable);
36     for (const auto& question : m_questions)
37     {
38         if (question.first == variable)
39         {
40             string answer { };
41             bool ok { false };
42             do
43             {
44                 cout << question.second << endl;
45                 getline(cin, answer);
46                 answer = toupper(answer);
47                 if (kb.isLegalAnswer(answer))
48                 {
49                     kb.setVariable(question.first, answer);
50                     kb.report("It has been learned that "s + question.first +
51                             "\nis "s + answer + "."s);
52                     ok = true;
53                 }
54             } else
55             {
56                 cout << "I'm sorry, but your answer is not acceptable.\n"s
57                 << "Please be sure you are typing the answer you "s
58                 << "want fully and correctly,\n"s
59                 << "and please choose your answer from one of these:"s
60                 << endl;
61                 kb.outputLegalAnswers();
62             }
63         } while (!ok);

```

```

64         return true;
65     }
66 }
67 return false;
68 }

```

Listing 166: *EC.cpp* – Questions class

Questions::size() simply forwards the call to vector<>::size() and returns its result.

Questions::add() adds a question consisting of variable and topic, both passed as parameters. The range-based for loop checks if a question for variable already exists in Questions::m_questions. If this is the case, the loop and method are exited with false. Otherwise, the question is added after the loop ends with m_questions.push_back(make_pair(variable,subject));. After that, the method returns true. Since Questions::add() returns a boolean value, the caller can check whether the question could be added or not. The method has no [[nodiscard]] attribute. Therefore its return value can be ignored.

In the Questions Section, it was analyzed how *ESIE*[™] behaves when there are multiple questions for a variable and the number of questions exceeds the documented limit (requirements 1.7.6, 1.7.7 and 2.1.17). It must be clearly stated that the above implementation of Questions::add() violates requirement 1.7.6. Obviously, requirement 1.7.6 does not make sense. This creates a conflict. There are three ways to resolve this:

1. Modify the implementation of Questions::add() to match requirement 1.7.6.
2. Modify requirement 1.7.6 to match the implementation of Questions::add().
3. Keep requirement 1.7.6 and the implementation of Questions::add(), document the deviation and resolve it in a future version of *ec/EC.cpp*.

The first option is easily implemented by the definition of Questions::add() shown in Listing 167.

```

1 bool Questions::add(const string& variable,const string& subject)
2 {
3     m_questions.push_back(make_pair(variable,subject));
4     return true;
5 }

```

Listing 167: *Questions::add()*, which fully complies with requirement 1.7.6

The second option is also easy to perform, as Table 27 shows. The requirement identifier is changed from 1.7.6 to 1.7.6r to indicate that it is a revised version. The text of the previous version is formatted in *italics and highlighted in light blue*.

In principle, a knowledge base with multiple questions for the same variable should be considered inconsistent. In practice, at least a warning should be issued to inform the user.

ID	Topic	Subtopic	Description	Source
1.7.6r	<i>Syntax</i>	<i>QUESTION</i>	Although <i>ESIE™</i> records questions without checking for complete duplicates or multiple but different questions, the prototype only accepts a maximum of one question per variable. If there are multiple questions for a variable, only the first of them is recorded.	<i>ESIE™</i> , <i>100SQ.KB</i>

Table 27: Requirement 1.7.6 revised

The third option does not seem attractive. Why not just decide between the first two options?

However, there is requirement 1.7.7, which states that – if there are multiple questions for a variable – they are all asked in the order in which they appear in the knowledge base. This has the consequence that the user can enter a different answer to the question each time. Only the last answer is relevant! Querying a knowledge base in this way is also inconsistent. Interestingly, the implementation of *ec/EC.cpp*, as it is, does not show the same behavior. As soon as a value is assigned to a variable, a subsequent assignment is no longer possible, in particular an assignment of another value does not take place. It can be speculated that the stack for tracking rules that assign a value to a particular variable and questions that ask for the value of a particular variable (requirement 3.3.10) is the basis for realizing this behavior. However, it was already decided not to implement a stack. It was replaced by the implicitly built function call stack in recursive and indirect recursive function calls. So following the first option consistently would mean further changes to the actual program. But choosing the second option also has consequences. Fixing inconsistent or erroneous requirements should not be done in isolation. Therefore, under the second option, it would make sense to add another requirement for reporting multiple occurrences of questions related to the same variable. In addition, the inference process should be cleaned up. This would also clarify what other deficiencies may exist with regard to the inference process and how they can be remedied.

Choosing one of the first two options means either living with more bugs or investing a lot more work to fix everything, depending on the choice made. In this way, the completion of the work on the first cover version of *ESIE™* would be delayed. For this reason, the third option is chosen here.

`Questions::output()` outputs the questions to a text file in a format valid for a knowledge base.

The member function `Questions::ask()` has two parameters, a reference to a `KnowledgeBase` named `kb` and a reference to a `const string` named `variable`. Its purpose is to find a question related to `variable` and ask it. When a related question is found and asked, the knowledge base is asked to assign the answer to the variable by `kb.setVariable(question.first, answer);`. After that the method returns `true`.

If no question for `variable` can be found, `false` is returned. This is the rough sketch of what happens in this method.

To search for a question for `variable`, a range-based for loop is used. To access the individual questions as values, the `variable` question is used. If `question.first` equals `variable`, a matching question is found. In this case, the `variable` answer of type `std::string` is declared in the corresponding compound statement. Next, the `bool` variable `ok` is declared and initialized with `false`. Now a `do` loop is started, which is executed as long as `ok` is `false`, that is, `while (!ok)`. In the compound statement of the `do` loop, the *subject*, i.e. `question.second`, is sent to the output and the user's answer is determined by calling `getline(cin, answer);`. `std::getline()` is a free function that takes an input stream as the first argument and a variable of type `std::string` as the second argument. There is a third argument to pass a delimiter character that signals the end of the input. It has the default value `'\n'`.

`getline()` reads characters from the input stream, here `cin`, and stores them in the string variable passed by reference, here `answer`, as long as the delimiter is not read, here `'\n'`. Character extraction stops when the delimiter is read or the end of the file is reached. The delimiter is extracted, but not stored in the string reference. The `getline()` function has a big advantage. It is not possible to provoke a buffer overflow by entering so many characters that the storage capacity of the string reference passed as parameter is exceeded.

Since the user can also enter lowercase letters, any lowercase letters contained in the response must be converted to uppercase. Then it must be checked whether the user has entered a legal answer. This is delegated to a method of the `KnowledgeBase` by calling `kb.isLegalAnswer(answer)`. `Questions::add()` cannot check this itself because `Questions` does not have access to `LegalAnswers`. The only member variable of `LegalAnswers` is managed exclusively by `KnowledgeBase`. If the answer is legal, it must be assigned to the variable. As mentioned earlier, this is done by `kb.setVariable(question.first, answer);`. After that, a yet unknown method of `KnowledgeBase` is called, namely `kb.report()`. If *TRACE ON* is entered before a consultation starts, `KnowledgeBase::report()` will output the appropriate tracing messages. After that, `ok` is set to `true`, which ends the execution of the `do` loop. If the answer given by the user is not legal, an appropriate message will be sent to output. By calling `kb.outputLegalAnswers()` the user will be informed about the legal answers.

Incidentally, a simple way to mimic the actual behavior of *ESIE*[™] when asking questions (requirement 1.7.7) would be to not end the for loop and method by returning `true`. Nevertheless, it should be recorded whether at least one question was asked, because only in this case the method should return `true`.

6.5.8. Rule Class

Rule is a domain-specific class that implements the domain concept *IF rule*. Listing 168 shows its definition as well as the implementation of `Rule::prove()`, which is defined outside the class after the definition of `KnowledgeBase`.

```
1 class Rule
2 {
3     public:
4         bool addCondition(const string& variable,const string& value)
5         {
6             for (const auto& element : m_conditions)
7             {
8                 if (element.first == variable)
9                 {
10                    return false;
11                }
12            }
13            m_conditions.push_back(make_pair(variable,value));
14            return true;
15        }
16        bool addConclusion(const string& variable,const string& value)
17        {
18            if (m_variable != ""s || m_value != ""s)
19            {
20                return false;
21            }
22            m_variable = variable;
23            m_value = value;
24            return true;
25        }
26        void output(ostream& os) const
27        {
28            // output if-part
29            os << "IF"s;
30            container_t::size_type count { 0 };
31            for (const auto& element : m_conditions)
32            {
33                os << " "s << element.first << " IS "s << element.second;
34                if (++count < m_conditions.size())
35                {
36                    os << "\nand"s;
37                }
38            }
39            // output then-part
40            os << "\nthen "s << m_variable << " is "s << m_value << '\n' << endl;
41        }
42        [[nodiscard]] bool isActive() const
43        {
44            return m_active;
45        }
46        void reset()
47        {
48            m_active = true;
49        }
50        bool prove(KnowledgeBase& kb,const string& variable);
51    private:
52        using container_t = vector<pair<string,string>>;
53        container_t m_conditions;
54        string m_variable { },
55            m_value { };
56        bool m_active { true };
57 };
58 // ...
```

```

59 bool Rule::prove(KnowledgeBase& kb,const string& variable)
60 {
61     Logger log("Rule::prove"s,variable);
62     if (variable != m_variable)
63     {
64         return false;
65     }
66     for (const auto& condition : m_conditions)
67     {
68         string value { kb.getValue(condition.first) };
69         if (value == ""s)
70         {
71             if (!kb.prove(condition.first) && !kb.askValue(condition.first))
72             {
73                 return false;
74             }
75             value = kb.getValue(condition.first);
76             kb.report("Currently looking for: "s + variable + "."s);
77         }
78         if (value != condition.second)
79         {
80             return false;
81         }
82     }
83     kb.setVariable(m_variable,m_value);
84     kb.report("Currently looking for: "s + variable + "."s);
85     kb.report("It has been learned that "s + m_variable +
86             "\nis "s + m_value + '.');
87     m_active = false;
88     return true;
89 }

```

Listing 168: `ec/EC.cpp` – Rule class

It introduces a new language feature, namely the *using declaration*. In the private part of the class, `using container_t = vector<pair<string,string>>;` declares `container_t` as an alias for the type `vector<pair<string,string>>`. A type alias is just another name for a type, much like a reference to a variable is just another name for that variable. The alias and the original are indistinguishable. They are identical in every respect. The scheme of a using declaration is `using alias = original;`. The *alias* is always to the left of `=` and the *original* is to the right of it. Why is it done this way now? Previously, any member variable that was a container was defined with the original type name. In these earlier cases, the type name was always used only once. But in `Rule`, the type name is needed a second time, in `Rule::output()`. Here the member type `vector<>::size_type` is used to define the local variable `count`. If the same complicated type name `vector<pair<string,string>>` had to be used repeatedly, this would be a potential source of errors. If, for whatever reason, `std::vector<>` were replaced with a different container template, this change would have to be applied consistently to every use of the container type. It is a common mistake that these necessary changes are not made everywhere. By introducing a type alias and using it consistently, this type of error is completely eliminated. Of course, one could argue that `std::intmax_t` could have been used here instead of `container_t::size_type`, but accessing the correct member type is always more appropriate.

The type alias `Rule::container_t` is used to declare `Rule::m_conditions`. This member variable manages all the conditions that a rule has. A rule must have at least one condition and may have many more as long as the total number of rule lines is not exceeded (requirements 1.6.5, 1.6.10 and 3.2.1). The conclusion of a rule consists of the *name* of a variable, `Rule::m_variable`, and a *value*, `Rule::m_value`, both of type `std::string`. If all the conditions of a rule evaluate to true, the rule is *fired*, that is, it executes its conclusion, which assigns the value to the variable. Again, a domain-specific concept, namely the *conclusion*, has no explicit counterpart in the implementation because it is too lightweight and there are already adequate means in the solution space. `Rule::m_active` is not motivated by a specific requirement. It was introduced after several knowledge bases had been processed with logging turned on. This showed that rules that had already been fired could be re-evaluated when processing the knowledge base. Since the values of the variables they refer to in the condition cannot be changed in the meantime, they are even fired again. So the actual overall state of the knowledge base is not changed by this. Still, it is a waste of time, and there is no guarantee that changes to the inference process, other components, or methods could change this. Therefore, this flag is set to false once a rule has been triggered. Before evaluating the conditions of a rule, it is first checked whether the rule is still active, i.e. it must be checked whether `Rule::isActive()` returns true. If the rule is no longer active, it is excluded from further processing.

The `Rule::addCondition()` method takes the *variable* and *value* parameters passed as a reference to const of type `std::string` and stores them as an (elementary) condition. If there is already a condition that uses the same *variable*, the method returns false, otherwise true. The implementation of the method should be easy to understand by now. Therefore, it will not be explained further. The method can be called multiple times while loading an *IF rule*, depending on the actual number of conditions.

The member function `Rule::addConclusion()` adds the parameters *variable* and *value* as a conclusion. If either `Rule::m_variable` or `Rule::m_value` is already set, i.e. has a value other than the empty string, the method returns false, otherwise true. This method should be called exactly once when loading a rule from the knowledge base.

The purpose of `Rule::output()` is to write a representation of the rule to a text file using a valid knowledge base format. This will output all conditions as well as the conclusion. An interesting implementation detail is how the local *variable count* is used to determine in the range-based for loop whether to send `"\nand"` to os before sending the next condition to os. The challenge is that there must not be an `"AND"` after the output of the last condition.

`Rule::isActive()` allows to query the status of the rule. It returns `Rule::m_active`. Its return value is prefixed with the `[[nodiscard]]` attribute, because it would be nonsensical to call this method and ignore its return value.

The `Rule::reset()` method is called after a consultation has ended to set `Rule::m_active` back to `true`, preparing the rule for an optional next consultation.

`Rule::prove()` is an important part of the knowledge base processing. Its parameters are a reference to the `KnowledgeBase` named `kb` and a reference to `const` of type `std::string` named `variable`. The knowledge base must be passed as a parameter because a rule does not have access to the exemplar of `Variables` stored in `kb`. It also asks `kb` to prove a variable or to ask for the value of a variable to assign values to an unassigned variable in one of its conditions. It also calls `KnowledgeBase::report()` to output the desired tracing messages if *TRACING ON* is set at the top level.

The following is a step-by-step description of the implementation of `Rule::prove()`.

The first statement creates an exemplar of `Logger` named `log`. Since `Rule::prove()` can call itself recursively, it can be very instructive to turn on logging and observe the stack trace of method calls to process the knowledge base. As mentioned earlier, this was essential to identify the right places to report the appropriate tracing messages.

Then it is checked whether the rule's conclusion concerns `variable`. If not, the method returns `false`. Otherwise, a range-based for loop is started. First, `kb` is asked for the value of `variable`. If its value is equal to the empty string, it means that the variable has not been assigned yet. In this case, an attempt is made to assign a value to it, either by calling `KnowledgeBase::prove()` or by calling `KnowledgeBase::askValue()`. The corresponding if condition is formulated in a short and concise way, namely `if (!kb.prove(condition.first) && !kb.askValue(condition.first))`, making use of short-circuit evaluation one more time. This will be explained in detail below.

`kb.prove(condition.first)` returns `true` if the variable referenced by `condition.first` was successfully assigned. In this case, the entire if condition immediately becomes `false`. This is because `true` is negated by the `!` operator, which results in `false`. Due to the short-circuit evaluation of the logical `And`, the remaining part is not evaluated because the overall result is already known to be `false`. If `kb.prove(condition.first)` fails, `kb.askValue(condition.first)` is evaluated. If it returns `true`, the negation returns `false`. Thus, the entire condition evaluates to `false`. If both condition parts evaluate to `true`, i.e. both method calls return `false` to indicate that the variable cannot be assigned by either proving or asking, the range-based for loop and `Rule::prove()` are exited by returning `false`.

If either proving or asking was successful, the actual value of the variable must be retrieved. This is done by `value = kb.getValue(condition.first);`. Then a corresponding tracing message is sent to the output if *TRACE ON* was set at the top level. Now it must be checked if the actual value of the variable matches the expected value in `condition.second`. If not, the for loop and the method are exited returning `false`. This is also an example of short-circuit evaluation of a condition with parts connected by logical And. Finally, the for loop is terminated. This means that every condition was evaluated as `true`. Thus, the entire condition is `true`, and the rule can *fire*, i.e., it executes its conclusion. Despite the somewhat dramatic wording, this is simply a request to the knowledge base to set the variable to a value by `kb.setVariable(m_variable,m_value);`. Again, two tracing messages are reported when *TRACE ON* was entered at the top level. The last action is to set `Rule::m_active` to `false`. This prevents the rule from being evaluated again during the active consultation.

6.5.9. Rules Class

The Rules class manages the *IF rules* contained in a knowledge base. Listing 169 also contains the definition of `Rules::prove()`, which is located in the source file after the definition of the KnowledgeBase class.

```

1 class Rules
2 {
3     public:
4         [[nodiscard]] size_t size() const
5         {
6             return m_rules.size();
7         }
8         // No check for verbatim or semantic duplicates!
9         void add(const Rule& rule)
10        {
11            m_rules.push_back(rule);
12        }
13        void output(ostream& os) const
14        {
15            for (const auto& element : m_rules)
16            {
17                element.output(os);
18            }
19        }
20        void reset()
21        {
22            for (auto& element : m_rules)
23            {
24                element.reset();
25            }
26        }
27        bool prove(KnowledgeBase& kb,const string& variable);
28    private:
29        vector<Rule> m_rules;
30 };
31 // ...
32 bool Rules::prove(KnowledgeBase& kb,const string& variable)
33 {
34     Logger log("Rules::prove"s,variable);

```

```

35     kb.report("Currently looking for: "s + variable + "."s);
36     for (auto& rule : m_rules)
37     {
38         if (rule.isActive())
39         {
40             if (rule.prove(kb,variable))
41             {
42                 return true;
43             }
44         }
45     }
46     return false;
47 }

```

Listing 169: EC.cpp – Rules class

Its only data member is `Rules::m_rules`, which is of type `vector<Rule>`. The `Rules::size()` method simply passes this request to `Rules::m_rules.size()`. `Rules::add()` adds a rule that is passed as a reference to `const` parameter of type `Rule`. No check is made to see if a semantically equivalent rule already exists. Therefore, the rule is added as the last element to `Rules::m_rules` by `m_rules.push_back(rule);`. Since no check is performed, this method returns nothing, i.e., its return type is `void`.

`Rules::output()` uses a range-based for loop to ask each rule to send itself to the `os` text file.

`Rules::reset()` uses a range-based for loop to ask each rule to reset itself, i.e. set its `Rule::m_active` data element to `true`.

The member function `Rules::prove()` has two parameters. The first is a reference to `KnowledgeBase` named `kb`, and the second is a reference to `const` of type `std::string` named `variable`. First, an exemplar of `Logger` named `log` is declared. If logging is turned on, the call to and exit from `KnowledgeBase::prove()` is logged. Next, a tracing message is emitted if `TRACE ON` is set. After that, a range-based for loop is started. If the rule referenced by the loop variable is active, it is requested with `rule.prove(kb,variable)` to assign a value to `variable`. If this is successful, the loop and the method are exited with `return true;`. If no rule was found that assigns a value to `variable`, the method returns `false`. Since the return value of type `bool` is not attributed with `[[nodiscard]]`, the caller can ignore it.

6.5.10. KnowledgeBase Class

`KnowledgeBase` is the central class and with almost 400 lines makes up about half of the entire program. For this reason, there is no complete listing of this class. Its data members and methods are presented in various listings. Listing 170 shows the data members of `KnowledgeBase`. As usual, they are all declared as `private`.

```

1 class KnowledgeBase
2 {
3     public:

```



```

4 // ...
5 private:
6     string m_goal;
7     pair<string,string> m_answer;
8     LegalAnswers m_legalAnswers;
9     Rules m_rules;
10    Questions m_questions;
11    Variables m_variables;
12    bool m_tracing { false };
13    size_t m_ruleLines { 0 };
14 };

```

Listing 170: *EC.cpp* – Data members of *KnowledgeBase* class

The next data element, `KnowledgeBase::m_answer`, is of type `pair<string, string>`. It implements the second rule type mentioned in requirement 1.2.50, namely *ANSWER*. An *ANSWER* consists of a text and a variable. Of course, both components could have been declared as separate data elements, for example `string m_answerText` and `string m_answerVariable`. Combining them in a `std::pair<>` emphasizes their cohesion. On the other hand, this has the effect of mapping the domain-specific concepts *GOAL text* and *GOAL variable* to the unintentional implementation-specific names `m_answer.first` and `m_answer.second`. If relevant coding guidelines do not provide guidance on how to decide this, it is a matter of taste which option is chosen. However, regardless of the actual decision, similar cases should be decided similarly.

The third data member, `KnowledgeBase::m_legalAnswers` of type `LegalAnswers`, corresponds to the third rule type mentioned in requirement 1.2.50, namely *LEGALANSWERS rule*.

The fourth data member, `KnowledgeBase::m_rules` of type `Rules`, is not a direct equivalent of the *IF-Rule* (requirement 1.2.50), but manages the rules contained in a knowledge base. A data structure for managing *IF-Rules* is not explicitly mentioned in the requirements.

The same is true for the fifth data member, `KnowledgeBase::m_questions` of type `Questions`, which is also not a direct equivalent of the *QUESTION rule* (requirement 1.2.20), since it manages questions contained in a knowledge base.

The domain-specific concept *variable* is frequently mentioned in the requirements, but there is no explicit requirement for a data structure to manage variables, which is the purpose of `KnowledgeBase::m_variables` of type `Variables`.

The data member `KnowledgeBase::m_tracing` of type `bool` contains the information whether tracing is enabled. According to requirement 2.2.15 it is initialized to `false` in its declaration.

`KnowledgeBase::m_ruleLines` is of type `size_t` and is initialized to 0 in its declaration. It is the implementation counterpart to the domain-specific concept *rule line* that occurs in requirements 1.6.5 and 1.6.10.

6.5.10.1. KnowledgeBase Member Functions

KnowledgeBase member functions can be roughly divided into five categories:

1. *Loading* – The member functions in this category implement loading the contents of a knowledge base. It includes a helper function, `KnowledgeBase::error()`, and a set of member functions whose names begin with `input`.
2. *Testing* – This category contains only one member function, `KnowledgeBase::output()`. It requests all data members that contain parts of a knowledge base to output them to a text file in a valid knowledge base format.
3. *Processing* – This category also has only one function, namely `KnowledgeBase::prove()`, which serves as the entry point of the inference process.
4. *Interaction* – All member functions that involve interaction with the user are included in this category, namely `KnowledgeBase::run()`, `KnowledgeBase::inputCommand()` and `KnowledgeBase::report()`.
5. *Forwarding* – This category includes methods that simply forward external requests to KnowledgeBase to its data members. It is considered bad programming style to expose data members of a class to external clients. To avoid this, a class must provide methods that forward external messages to the appropriate data members and return the results. Members of this category are `KnowledgeBase::isLegalAnswer()`, `KnowledgeBase::outputLegalAnswers()`, `KnowledgeBase::setVariable()`, `KnowledgeBase::getValue()`, and `KnowledgeBase::askValue()`.

The following listings are excerpts from the definition of the KnowledgeBase class. The declarations of the methods as well as their definitions are all included in the public part of the KnowledgeBase class. The surrounding parts of the class definition are not included. In addition, the definitions of the methods in the source file are indented accordingly. However, in the following listings, the indentation relative to the class definition has been removed.

6.5.10.2. Member Functions for Loading

Listing 171 shows the `KnowledgeBase::error()` helper function. It accepts a reference to `const` of type `std::string` named `message`. It outputs `message` to `cerr`, the global unbuffered error stream. Since everything sent to `cerr` is output immediately without buffering, the error message is not lost if, for example, the program terminates abnormally due to the reported error.

It also returns `false` as a result. So it can be called to report an error, and it can also be returned as a result by the calling function. Its use and utility are best illustrated

with an example. The code fragment in Listing 172 checks for an error, prints an error message, and exits the function returning false.

```
1 bool error(const string& message) const
2 {
3     cerr << message << endl;
4     return false;
5 }
```

Listing 171: *EC.cpp – KnowledgeBase::error()*

```
1 if (m_legalAnswers.isActive())
2 {
3     cerr << ("LEGALANSWERS have been specified more than once "s +
4             "in the knowledge base."s)
5         << endl;
6     return false
7 }
```

Listing 172: *Checking for and reporting an error*

Using KnowledgeBase::error(), the code fragment from Listing 172 can be rewritten into a more concise form, as Listing 173 shows.

```
1 if (m_legalAnswers.isActive())
2 {
3     return error("LEGALANSWERS have been specified more than once "s +
4                 "in the knowledge base."s);
5 }
```

Listing 173: *Checking for and reporting an error with KnowledgeBase::error()*

Since the return value of KnowledgeBase::error() is not prefixed with `[[nodiscard]]`, its return value can be ignored. In this case, only the error message is sent to `cerr`.

Next, the various methods whose names begin with `input` are introduced. Together they form a *parser* that parses the syntactic elements of a knowledge base, namely the five rule types including their elements, and finally loads and stores them. The parsing process is organized hierarchically. It starts with the call to KnowledgeBase::input(), which repeatedly calls KnowledgeBase::inputToken(). Depending on the extracted token, parsing continues by calling another input method, for example KnowledgeBase::inputLegalAnswers(). The latter usually calls KnowledgeBase::inputToken() to parse the remaining parts. In the following presentation, a reverse order is used. The lowest level input method, namely KnowledgeBase::inputToken(), is presented first, while the top level input method, namely KnowledgeBase::input(), is presented last.

Listing 174 shows the definition of the member function KnowledgeBase::inputToken().

```
1 [[nodiscard]] string inputToken(istream& is) const
2 {
3     string token;
4     is >> ws;
5     if (!is)
```

```

6   {
7   error("Unexpected end of file encountered in rule file."s);
8   }
9   if (is.peek() == '\\")
10  {
11  is.get(); // discard initial quote
12  while (is && is.peek() != '\\")
13  {
14  token += is.get();
15  }
16  if (is && is.peek() == '\\")
17  {
18  is.get(); // discard closing quote
19  }
20  if (token.length() > 80)
21  {
22  token = token.substr(0,80);
23  }
24  }
25  else
26  {
27  while (is && is.peek() > 32 && is.peek() != '\\")
28  {
29  token += toupper(is.get());
30  }
31  if (token.length() > 40)
32  {
33  token = token.substr(0,40);
34  };
35  };
36  return token;
37 }

```

Listing 174: EC.cpp – KnowledgeBase::inputToken()

It is largely identical to the inputToken() function in Listing 145. This function inputs either a double-quoted string containing all characters except double quotes, or a string of consecutive characters without spaces. So only the differences are explained.

- The first difference extends over lines 5 – 8 and introduces an additional check whether the input text stream is still OK. If not, a corresponding error message is output.
- The second difference, which spans lines 20 – 23, relates to the if branch that is executed when a double-quoted string is entered. A string in double quotes conforms to the domain-specific concept *text* (requirement 1.2.40). According to requirement 1.2.45, it may need to be truncated to a maximum length of 80 characters.
- The third difference, lines 31 – 34, is in the else branch, where a string of consecutive non-blank characters is entered. Such a string corresponds to the domain-specific terms *variable* and *value*. According to requirement 1.2.35, it will be shortened to a maximum of 40 characters, if necessary.

The return value is prefixed with the [[nodiscard]] attribute because it should not be ignored.

The syntax of all rules includes the use of *is* or *are*, e.g., requirements 1.3.0, 1.4.0, 1.5.0, 1.6.0, 1.7.0. Listing 175 illustrates how to parse this syntax element.

```
1 bool inputIsAre(istream& is) const
2 {
3     string isAre { inputToken(is) };
4     if (!is && isAre == "")
5     {
6         return false;
7     }
8     /* // with checking:
9     if (isAre == "IS"s || isAre == "ARE"s)
10    {
11        return true;
12    }
13    else
14    {
15        return false;
16    }
17    */
18    return true;
19 }
```

Listing 175: *EC.cpp* – *KnowledgeBase::inputIsAre()*

Playing around with different knowledge bases, it turned out that any token can occur instead of "is" or "are". Therefore, the check for the presence of "is" or "are" was turned into a comment to achieve the same behavior as *ESIE*[™], which does not conform to its specification in this respect.

Attention should be paid to cases in which there are no more tokens at all, i.e. the end of file has been reached, or an empty token was read. The latter can happen if a pair of immediately consecutive quotes is used instead of "if".

In a future version it would be advisable to use a strict syntax checking in this case and report any errors to the user.

The member function returns a `bool` value indicating whether the desired syntax element was successfully extracted or not. The actual implementation will always return `true`. Since there is no `[[nodiscard]]` attribute, the return value can be ignored by the caller.

Listing 176 shows the member function that extracts a variable/value pair connected by "is" or "are". The schema is `<variable> is <value>`.

```
1 bool inputVariableValue(istream& is, string& variable, string& value) const
2 {
3     // input variable
4     variable = inputToken(is);
5     if (!is && variable == "")
6     {
7         return false;
8     }
9     // input is/are
10    if (!inputIsAre(is))
11    {
12        return false;
13    }
```

```

14 // input value
15 value = inputToken(is);
16 if (!is && value == "")
17 {
18     return false;
19 }
20 return true;
21 }

```

Listing 176: *EC.cpp – KnowledgeBase::inputVariableValue()*

The reference parameters `variable` and `value`, both of type `std::string`, contain the parsed elements if the method returns `true`. Otherwise, the contents of the references are not defined.

First, `KnowledgeBase::inputToken()` is called and the result is assigned to `variable`. If the input stream is empty or an empty string was read, the method is exited with `return false;`. It should be mentioned again that an empty string results from parsing a string that consists of two consecutive double quotes.

Next, "is" or "are" must be read. To do this, the member function `KnowledgeBase::inputIsAre()` is called. If the incorrect implementation of `KnowledgeBase::inputIsAre()` is corrected sometime in the future, nothing in `KnowledgeBase::inputToken()` needs to be changed. This can be considered good design.

Then `value` is parsed in exactly the same way. So there is no need to declare it again. Again, the return value of the method indicating success or failure can be ignored by the caller.

Listing 177 shows the member function for parsing the *LEGALANSWERS* rule.

```

1 bool inputLegalAnswers(istream& is)
2 {
3     if (m_legalAnswers.isActive())
4     {
5         return error("LEGALANSWERS have been specified more than once "s +
6                     "in the knowledge base."s);
7     }
8     m_legalAnswers.setActive();
9     // input is/are
10    if (!inputIsAre(is))
11    {
12        return false;
13    }
14    // input legal answers including terminator *
15    string answer { };
16    do
17    {
18        answer = inputToken(is);
19        if (!is && answer == "")
20        {
21            return false;
22        }
23        if (answer != "*"s)
24        {
25            m_legalAnswers.add(answer);
26            if (m_legalAnswers.size() > 50)

```

```

27     {
28         return error("Too many legalanswers encountered in the "s +
29                     "LEGALANSWERS rule."s);
30     }
31 };
32 } while (answer != "*"s);
33 return true;
34 }

```

Listing 177: *EC.cpp – KnowledgeBase::inputLegalAnswers()*

First, the method checks if legal answers have already been parsed (line 3). In this case, `m_legalAnswers.isActive()` returns true. If this is the case, an error message is issued and the method exits with the return value of `KnowledgeBase::error()` (lines 5 – 6), which is always false. Next, `m_legalAnswers.setActive()` is called to record that the only instance of *LEGALANSWERS rule* is parsed (requirement 1.5.0). Then "is" or "are" must be extracted by calling the appropriate method. If this fails, the method returns false. Then it checks if the end of the file has been reached or an empty string was read. Now a do loop starts (line 16). It reads the next token by calling `KnowledgeBase::inputToken()`. Again it checks if the end of the file is reached or if there is an empty string. Then it checks if the token is different from "*" (line 23), because this particular token terminates the *LEGALANSWERS rule* (requirement 1.5.0). If it is different from "*", the answer was successfully read and is stored with `m_legalAnswers.add(answer);`. Based on requirement 1.5.5, it then checks if there are more than 50 answers. If so, `KnowledgeBase::error()` reports an error and the method returns false. The condition of the do loop (line 32) evaluates to true if the response is different from "*", that is, if the terminating token was not extracted. Finally, the method returns true to indicate success.

```

1 bool inputGoal(istream& is)
2 {
3     if (m_goal != ""s)
4     {
5         return error("Goal encountered more than once in "s +
6                     "Knowledge Base."s);
7     }
8     if (!inputIsAre(is))
9     {
10        return false;
11    }
12    m_goal = inputToken(is);
13    if (!is && m_goal == ""s)
14    {
15        return false;
16    }
17    m_variables.add(m_goal);
18    return true;
19 }

```

Listing 178: *EC.cpp – KnowledgeBase::inputGoal()*

Listing 178 shows the member function `KnowledgeBase::inputGoal()`. Its implementation follows the scheme of `KnowledgeBase::inputLegalAnswers()`. Detection of whether a *GOAL rule* is processed more than once is done by checking

whether `m_goal` is different from the empty string. Thus, no separate data element is required to track this.

The member function for reading an *IF rule* is more extensive and complex (Listing 179). It must be mentioned again that it is called `inputRule()` and not `inputIf()`, since the *IF rule* is the only *rule* recognized in classical expert system terminology, for example (*OECD Glossary of Statistical Terms - Expert System Definition*, n.d.), (“Expert System,” 2021) and (“Rule-Based Machine Learning,” 2021).

```
20 bool inputRule(istream& is)
21 {
22     string variable,
23         value,
24         token;
25     Rule rule;
26     // input if part
27     do
28     {
29         if (!inputVariableValue(is,variable,value))
30         {
31             return false;
32         }
33         m_variables.add(variable);
34         rule.addCondition(variable,value);
35         ++m_ruleLines;
36         if (m_ruleLines >= 400)
37         {
38             return error("There are too many rules in the Knowledge Base "s +
39                         "for me."s);
40         }
41         token = inputToken(is);
42         if (!is && token == "s)
43         {
44             return false;
45         }
46     } while (token == "AND"s);
47
48     // input then part
49     // if (token == "THEN") // ESIE accepts any token here; found "the "
50     // instead of "then " in "ANIMAL"
51     if (token != "s)
52     {
53         if (!inputVariableValue(is,variable,value))
54         {
55             return false;
56         }
57         m_variables.add(variable);
58         ++m_ruleLines;
59         rule.addConclusion(variable,value);
60         m_rules.add(rule);
61         return true;
62     }
63     return false;
64 }
```

Listing 179: *EC.cpp* – *KnowledgeBase::inputRule()*

It consists of two parts, each preceded by a corresponding comment. The first part (lines 26 – 46) parses the *if part*, i.e. the (*complex*) *condition*, which must consist of at least one (*elementary*) *condition*. The second part (lines 48 – 56) parses the

conclusion. Four local variables are declared. *variable*, *value* and *token* are of type `std::string`, *rule* is of type `Rule`.

Since the condition can consist of more than one elementary condition, a do loop is used. It is executed as long as *token* equals "AND". First, the do loop calls `inputVariableValue(is,variable,value)`. If this method call returns false, the method returns false. Next, `m_variables.add(variable)`; adds *variable* to `KnowledgeBase::m_variables`. Then `rule.addCondition(variable,value)`; adds *variable* and *value* as a new condition to the rule. Now `++m_ruleLines`; increments the number of rule lines, and the following if statement checks if the limit of rule lines according to requirement 1.6.10 is reached. If this is the case, an error message is issued and the method returns false. Next, a new token is read and assigned to *token*. If *token* is different from "AND", the do loop ends and parsing of the *then part* begins.

Here, another deviation of *ESIE*[™] from its requirements, namely requirement 1.6.0, became apparent. The knowledge base named *ANIMAL* or *ANIMAL.ESI* was loaded successfully. However, exporting it by calling `KnowledgeBase::output()` resulted in a significantly smaller file. A cursory examination showed that rules were missing. A detailed analysis revealed that the rule in lines 351 – 363 caused the problem. Listing 180 shows this rule, numbered here from 1 to 3.

```
1 if type teeth is  
2 and large.ears is no  
3 the species.animal is rat/mouse/squirrel/beaver/porcupine.
```

Listing 180: ANIMAL knowledge base (excerpt)

The error is hard to spot. It is the first word in line 363, i.e. the third line of the excerpt. A human reads "*the kind of animal is*", but the parsing in `KnowledgeBase::inputRule()` continues until an exact "THEN" is found, skipping everything else in between. This was the first observation that in *ESIE*[™] some reserved words can deviate in syntax from their specification.

With this knowledge, the problem could be easily fixed. Now it is checked if a non-empty string was extracted. If so, the conclusion is read by calling `inputVariableValue(is,variable,value)`. Then *variable* must be added to the variable store by calling `m_variables.add(variable)`. Next, `m_ruleLines` must be incremented by one, and the conclusion must be added to rule by calling `rule.addConclusion(variable,value)`. Next, rule must be added to the rule store, which is done by calling `m_rules.add(rule)`. Then, the method ends by returning true. Finally, if no valid variable value pair was read, the method returns false.

Listing 181 shows the implementation of `KnowledgeBase::inputQuestion()`. By and large, it should be obvious how it works if one has read and understood the preceding descriptions of the input methods. Therefore, the method will not be explained in detail. Two points can be mentioned. First, the actual number of

questions is compared to the limit (line 3, requirement 1.7.5). At the end of the method, the variable is added to the variable store (line 25) and the question itself is added to the question store (line 26).

```

1 bool inputQuestion(istream& is)
2 {
3     if (m_questions.size() >= 100)
4     {
5         return error("There are too many questions in the "s +
6                     "Knowledge Base for me."s);
7     }
8     // input variable
9     string variable { inputToken(is) };
10    if (!is && variable == "s")
11    {
12        return false;
13    }
14    // input is/are
15    if (!inputIsAre(is))
16    {
17        return false;
18    }
19    // input subject
20    string subject { inputToken(is) };
21    if (!is && subject == "s")
22    {
23        return false;
24    }
25    m_variables.add(variable);
26    m_questions.add(variable,subject);
27    return true;
28 }

```

Listing 181: *EC.cpp – KnowledgeBase::inputQuestion()*

The member function `KnowledgeBase::inputAnswer()` in Listing 182 largely corresponds to `KnowledgeBase::inputQuestion()` (Listing 181). The check for whether the *ANSWER rule* occurs more than once is similar to that in `KnowledgeBase::inputGoal()` (Listing 178), but now for both parts of `KnowledgeBase::m_answer`. At the end, the variable part of answer is added to the variable store by `m_variables.add(variable);`. Then, subject and variable are assigned to `KnowledgeBase::m_answer` by `m_answer = make_pair(subject, variable);`.

```

1 bool inputAnswer(istream& is)
2 {
3     if (m_answer.first != "s" || m_answer.second != "s")
4     {
5         return error("Answer encountered more than once in "s +
6                     "Knowledge Base."s);
7     }
8     if (!inputIsAre(is))
9     {
10        return false;
11    }
12    string subject { inputToken(is) };
13    if (!is && subject == "s")
14    {
15        return false;
16    }
17    // input variable

```

```

18 string variable { inputToken(is) };
19 if (!is && variable == "")
20 {
21     return false;
22 }
23 m_variables.add(variable);
24 m_answer = make_pair(subject,variable);
25 return true;
26 }

```

Listing 182: EC.cpp – KnowledgeBase::inputAnswer()

```

1 bool input(istream& is)
2 {
3     while (is)
4     {
5         string token { inputToken(is) };
6         if (token == "LEGALANSWERS"s)
7         {
8             if (!inputLegalAnswers(is))
9             {
10                return false;
11            };
12        }
13        else if (token == "GOAL"s)
14        {
15            inputGoal(is);
16        }
17        else if (token == "IF"s)
18        {
19            inputRule(is);
20        }
21        else if (token == "QUESTION"s)
22        {
23            inputQuestion(is);
24        }
25        else if (token == "ANSWER"s)
26        {
27            inputAnswer(is);
28        }
29        else if (token != "")
30        {
31            error ("Invalid rule found in Knowledge Base.\n"s +
32                "Rule begins with: "s + token);
33        };
34    }
35    bool goalAndAnswerFound { true };
36    if (m_goal == "")
37    {
38        error ("Goal statement not found "s +
39            "in the Knowledge Base."s);
40        goalAndAnswerFound = false;
41    }
42    if (m_answer.first == "" && m_answer.second == "")
43    {
44        error ("Answer statement not found "s +
45            "in the Knowledge Base."s);
46        goalAndAnswerFound = false;
47    }
48    return goalAndAnswerFound;
49 }

```

Listing 183: EC.cpp – KnowledgeBase::input()

Listing 183 shows the last member of the input methods, namely `KnowledgeBase::input()`. As mentioned earlier, this method is the entry point for parsing and loading a knowledge base.

The method starts with a while loop, that runs as long as the text stream containing the knowledge base is in a valid state, i.e. it can be converted to the bool value `true`.

The first syntactic element of a knowledge base must be a rule, optionally preceded by whitespace. Each rule begins with a specific token that specifies the rule type. Therefore, the first action of the initial while loop of `KnowledgeBase::input()` is to initialize the variable `token` of type `std::string` with the result of the call to `KnowledgeBase::inputToken()`. Then, a sequence of if statements checks whether `token` corresponds to a rule name. If so, the corresponding input method is called to parse and load the remaining parts of the rule. Actually, only legal responses are checked to determine if this rule was successfully parsed and loaded. If this is the case, `KnowledgeBase::input()` returns `false`. The results of the following input methods are ignored. So if they return `false` due to errors, this is neither noticed nor reported. This should be changed in a future version of *EC*.

At the end of the while loop, a check is made to see whether a *GOAL rule* and an *ANSWER rule* are present. If one is missing, a corresponding error message is issued. It should be noted that in both cases the result of the call to `KnowledgeBase::error()` is not returned immediately. This would cause `KnowledgeBase::input()` to terminate prematurely. However, *ESIE™* is capable of reporting both errors when they occur. Therefore, the variable `goalOrAnswerFound` of type `bool` is necessary to keep track of whether the *GOAL rule*, the *ANSWER rule*, or both were not found. Consequently, `KnowledgeBase::input()` returns `goalOrAnswerFound` as the result.

One remark must be made. The syntactic structure of a knowledge base can be described by a formal grammar, just as C++ as a programming language can be described by a formal grammar. There are different formats for formal grammars, which are also standardized. If a formal grammar is available, it can be processed by a so-called *parser generator*. This takes a formal grammar as input and generates a program that parses a text according to the given formal grammar and converts it into a data structure that is kept in the memory of a program and can be further processed. There are various parser generators for C++ in the form of stand-alone tools or libraries.

In this case study, neither a formal grammar nor a parser generator was used, as this would have required a very comprehensive introduction.

6.5.10.3. Member Function for Testing

Listing 184 shows the member function that outputs a previously loaded knowledge base. Since a KnowledgeBase exemplar manages goal and answer itself, it is also responsible for sending the corresponding variables to a text stream. For the remaining rules, i.e. legal answers, if-rules, and questions, the corresponding member variables are asked to send their contents to the output by calling their corresponding member functions.

```
1 void output(ostream& os)
2 {
3     os << "goal is "s << m_goal << endl << endl;
4     os << "answer is "s << '\n' << m_answer.first << "\n "s
5         << m_answer.second << endl << endl;
6     m_legalAnswers.output(os);
7     os << endl;
8     m_rules.output(os);
9     os << endl;
10    m_questions.output(os);
11 }
```

Listing 184: EC.cpp – KnowledgeBase::output()

6.5.10.4. Member Function for Processing

Listing 185 shows the member function for processing a knowledge base.

```
1 bool prove(const string& variable)
2 {
3     Logger log("KnowledgeBase::prove"s, variable);
4     return m_rules.prove(*this, variable) ||
5         m_questions.ask(*this, variable);
6 }
```

Listing 185: EC.cpp – KnowledgeBase::prove()

First, an exemplar of Logger is created, which is used to print a call stack trace when the global variable loggingActive is initialized to true. Then, m_rules.prove() is called as the left operand of the logical Or operator ||. If this method successfully assigns a value to the passed argument variable, the member function immediately returns true. In this case, C++ does not evaluate the second operand of the logical Or operator because of the short-circuit evaluation. Only if m_rules.prove() returns false, the second operand is called, namely m_questions.ask(). If this call evaluates to true, the parameter variable has been successfully assigned a value by asking the user, and the member function returns true. Otherwise, the member function returns false to indicate that it could not assign a value to variable. Originally, the logical Or operator and its right operand were not present. It was assumed that ESIE™ would not ask for the target variable. None of the knowledge bases supplied with ESIE™ contained an example of this case. However, it later turned out that ESIE™ may very well ask for the target variable if it cannot prove it by applying the rules and questions only to variables

used in conditions. One might object that a user consults a knowledge base only when the solution to the given problem is not known, but there seems to be no formal reason preventing *ESIE*[™] from doing so.

6.5.10.5. Member Functions for Interaction

Listing 186 shows a helper function extracted from the member function `KnowledgeBase::run()`. It starts a do loop that runs until the user enters a valid command, namely *TRACE ON*, *TRACE OFF*, *GO*, or *EXIT*. If the command is invalid, an error message informs about it and lists valid commands. After entering a valid command, it is returned as a value of type `std::string`. So this member function is part of the user interface. As mentioned before, *ESIE*[™] runs in graphical mode in a *DOS* emulator. This feature is not mimicked by *EC*, as it is a console-only application. This allows input and output to be redirected, which can be useful for testing.

```

1 string inputCommand() const
2 {
3     string command;
4     bool isKnown { false };
5     do
6     {
7         cout << "==" << flush;
8         getline(cin,command);
9         command = toupper(command);
10        if (command == "TRACE ON"s || command == "TRACE OFF"s ||
11            command == "GO"s || command == "EXIT"s)
12        {
13            isKnown = true;
14        }
15        else
16        {
17            error ("I don't understand that command.\n\n"s +
18                "Valid options are: "s +
19                "TRACE ON, TRACE OFF, GO, and EXIT."s);
20        }
21    } while (isKnown == false);
22    return command;
23 }
```

Listing 186: *EC.cpp* – `KnowledgeBase::inputCommand()`

The member function `KnowledgeBase::run()` implements the core functionality of the user interface. It is shown in Listing 187.

```

1 void run()
2 {
3     string command;
4     do
5     {
6         command = inputCommand();
7         if (command == "TRACE ON"s && m_tracing == false)
8         {
9             m_tracing = true;
10            report("There were "s + to_string(m_ruleLines) +
11                " rule-lines, "s +
12                to_string(m_questions.size()) + " questions and "s +
13                to_string(m_legalAnswers.size()) + "\nlegal answers "s +
```

```

14         "specified in the knowledge base."s);
15     };
16     if (command == "TRACE OFF"s)
17     {
18         m_tracing = false;
19     };
20     if (command == "GO"s)
21     {
22         if (prove(m_goal))
23         {
24             cout << m_answer.first << m_variables.get(m_answer.second)
25                 << endl << endl;
26         }
27         else
28         {
29             cout << "Error in Knowledge Base.\n"s
30                 << m_goal << " searched for but not found.\n"s
31                 << m_answer.first << "UNKNOWN\n"s << endl;
32         }
33         cout << "I have completed this analysis."s << endl;
34         m_variables.reset();
35         m_rules.reset();
36     }
37 } while (command != "EXIT"s);
38 cout << "Have a nice day!"s << endl;
39 }

```

Listing 187: *EC.cpp* – *KnowledgeBase::run()*

It implements the top-level mode, which is executed until the user enters *EXIT* or an error occurs that terminates the program. After declaring the local variable `command` of type `std::string`, a do loop is executed as long as `command` is different from "EXIT"s. In the do loop, the user is prompted to enter a command, which is assigned to the local variable `command`. After that the *TRACE ON* or *TRACE OFF* commands are processed. Whether tracing is active is tracked by `KnowledgeBase::m_tracing`. If tracing is on, an immediately repeated command *TRACE ON* must not cause the first tracing report to be output again. Thus, the condition of the if statement first checks the command and also the current value of `KnowledgeBase::m_tracing`, i.e. if `(command == "TRACE ON"s && m_tracing == false)`. Only if the entire condition evaluates to true will `KnowledgeBase::m_tracing` be set to true and the first tracing report is output.

If `command` equals "TRACE OFF"s, `KnowledgeBase::m_tracing` is set to false. If `command` equals "GO"s, *EC* tries to prove the goal variable. If the call to `prove(m_goal)` returns true, the response is sent to `cout`. Otherwise, an appropriate error message is sent to `cout`. Both outputs are part of the regular control flow and are not caused by internal errors. Therefore, the corresponding messages are sent directly to `cout`. `KnowledgeBase::error()` is not used for this purpose! Afterwards a message is sent to `cout` informing about the end of the consultation.

At the end of the do loop, a closing formula is sent to `cout`.

What remains are the member function `KnowledgeBase::report()`, which is involved in tracing, and some other member functions that simply forward queries to data members of `KnowledgeBase`. They are all shown in Listing 188.

```
1 void report(const string& message) const
2 {
3     if (m_tracing)
4     {
5         cout << message << endl;
6     }
7 }
8 bool isLegalAnswer(const string& answer) const
9 {
10    return m_legalAnswers.isLegal(answer);
11 }
12 void outputLegalAnswers() const
13 {
14    m_legalAnswers.output();
15 }
16 bool setVariable(const string& variable, const string& value)
17 {
18    return m_variables.set(variable, value);
19 }
20 [[nodiscard]] string getValue(const string& variable) const
21 {
22    return m_variables.get(variable);
23 }
24 bool askValue(const string& variable)
25 {
26    return m_questions.ask(*this, variable);
27 }
```

Listing 188: `EC.cpp` – various

If `KnowledgeBase::m_tracing` is true, `KnowledgeBase::report()` sends a message to `cout`. It must be emphasized that this output is considered part of the user interface. Therefore, the `KnowledgeBase::error()` method is not used as it is considered part of the error handling. In this way, the user interface and error handling are effectively decoupled, which may facilitate future maintenance.

Each of the remaining methods calls a member function of a data member that actually executes the query. Where appropriate, the return value is prefixed with the `[[nodiscard]]` attribute. An alternative to designing these methods would have been to expose the corresponding data members, either directly by declaring them public, or through getter methods that return them. However, this would reveal `KnowledgeBase` internals. The consequences can be illustrated with an example: A later decision to manage variables in a completely different way may break client code that uses `KnowledgeBase::m_variables` directly. Therefore, the current design is preferable. As long as the interfaces, i.e. the prototypes of the corresponding methods, remain unchanged, the underlying implementation can be changed as desired.

Last but not least, Listing 189 shows the `main()` function. This function implements the interaction with the user that is not related to `KnowledgeBase::run()`, i.e. the *top level*. The `ifstream file { };` statement declares an input file stream object named

file. The name of its type `std::ifstream` is an acronym for *input file stream*. A do loop is then started. Within this, the variables `name` and `answer` are declared, both of type `std::string`. The free function `getline()` introduced above is used to input the file name. The next statement attempts to open the file by calling `file.open(name)`.

```

1 int main()
2 {
3     ifstream file { };
4     do
5     {
6         string name { },
7             answer { };
8         cout << "File name where rules found?: ";
9         getline(cin,name);
10        file.open(name);
11        if (!file)
12        {
13            cout << "File \"" << name << "\" does not exist. "s
14                << "Do you wish to try again? (Y/N)"s << flush;
15            do
16            {
17                getline(cin,answer);
18                answer = toupper(answer);
19            }
20            while (answer != "Y"s && answer != "N"s);
21            if (answer == "N"s)
22            {
23                return 0;
24            }
25        }
26    } while (!file);
27    KnowledgeBase kb;
28    if (kb.input(file))
29    {
30        kb.run();
31    }
32 }

```

Listing 189: `EC.cpp – main()`

If file could not be opened successfully, applying the not operator to `file` results in true. In this case, an appropriate error message is sent to `cout`. It should be noted that `KnowledgeBase::error()` is not and should not be used to report this error, since `KnowledgeBase::error()` is a member function of `KnowledgeBase` and is not accessible to `main()` at this point. Furthermore, this error can be considered part of the user interface since the program does not terminate but continues to interact with the user. Therefore, the error message is sent to `cout` and not to `cerr`. If opening file failed, a do loop asks the user if another attempt is desired. If so, the outer do loop continues as long as `!file` returns true. If `answer` equals "N", `main()` immediately returns 0, thus terminating the program regularly. If file was opened successfully, a `KnowledgeBase` named `kb` is declared. After that, an if statement checks whether `kb.input(file)` returns true. If this is the case, `kb.run()` is called, reaching the top level of *EC*.

6.6. Evaluation

Answering some questions will help to assess what has been achieved with the *EC* program so far. This will help to appreciate *EC* and the work that has gone into its implementation.

Is EC meeting its requirements?

The overall requirement for *EC* is to become a functionally equivalent implementation of *ESIE*[™] using C++ and a modern platform. Previously, this overall requirement was broken down into many detailed requirements. For most of them, it was discussed whether or not they are implemented correctly in *EC*. It became clear that *ESIE*[™] itself does not fully meet the requirements. In addition, some of the requirements are incorrect or inconsistent. So the answer is that *EC* largely complies with the requirements, but not completely. Now, one might ask whether this is bad or not. In the given context, this is not bad. *ESIE*[™] has been sufficiently analyzed to generate ideas for a re-implementation and to ensure that this re-implementation works. Therefore, working on the remaining and finest details will certainly not yield any relevant insights or benefits. Instead, it is plausible that more rigorous testing could find even more deviations from *EC* to *ESIE*[™]. However, from a practical standpoint, it is not economically justifiable to put even more work into this version of *EC* to better meet the requirements. As mentioned earlier, it is impossible to become completely correct, i.e., for *EC* to correctly implement each of its requirements without changing the requirements themselves, since they are neither complete, nor correct, nor consistent.

Can EC be considered a final product?

To a certain extent, yes. *ESIE*[™] was an end product in its time, and *EC* largely mimics *ESIE*[™] very well. Today, however, *ESIE*[™] and thus *EC* are mainly of historical or didactic interest. Several inference engines exist in the form of libraries that allow them to be integrated into other applications. So there is probably no need to implement another one for professional purposes.

Does EC exhibit good software characteristics?

Answering this question raises many technical issues. First, despite its object-oriented design and use of classes, *EC* is not well modularized. KnowledgeBase, for example, is a monster class that is difficult to read and understand. But it would be very hard to find a much better solution in the first step. Applying changes to *EC* can also be difficult. For example, changing the processing of the knowledge base would be a big problem because the processing is spread among different classes, namely KnowledgeBase, Rules, Rule and Questions. Another drawback is that *EC* is only minimally tested, especially not in an automated way. Moreover, not all possible

variations of syntactic and semantic details of knowledge bases have been tested, even compared to *ESIE*[™].

Furthermore, the source code of *EC* has not yet been inspected in detail. Such an inspection would require that an ideally independent expert critically examines whether the implementation can be improved in terms of proper modularization, reuse of existing functions, understandability, and extensibility – all within the limits imposed by the relevant requirements. For example, `KnowledgeBase::error()` can be considered poorly modularized. First, it requires an exemplar of `KnowledgeBase` to be called. This is currently not a problem, since it is only called by non-static member functions of `KnowledgeBase`, i.e., there is always an exemplar of `KnowledgeBase` when it is called. More appropriately, it would have to be implemented as a static member function.

In addition, it should be thoroughly evaluated whether it makes sense to provide further domain-specific equivalents for domain-specific concepts and to critically evaluate each current domain-specific implementation part whether it is really justified.

Last but not least, the current implementation of *EC* lacks integrated documentation, which is a serious quality deficiency in terms of understandability and maintenance.

All in all, the criticism may sound harsh, but it is not; in fact, it is justified. However, the achieved result is also reasonable in terms of the resources used, especially human labor. Although *EC* is a small application for an experienced software developer with high expertise, it is a significant effort for a beginner programmer.

6.7. Outlook

What would be reasonable next steps if more resources were available?

Based on the manual tests, *EC* appears to be working well. However, when changes are made to *EC*, these tests must be repeated manually to determine if the changes have undesirable effects. So the main obstacle is the lack of automated testing. If automated tests were available, more variations of subtle details could be checked. Therefore, providing comprehensive automated testing is a top priority before making further changes.

A next important step would be to clean up the implementation. There should be a clear separation between auxiliary functions and classes and classes related to knowledge processing. Problems, such as the aforementioned `KnowledgeBase::error()` member function, should be identified and fixed. In addition, all

functions and classes should be completely separated in header and implementation files. Suitable namespaces should also be introduced.

Code-centric documentation for *EC* should be added by now at the latest. It is to be hoped that important information on the various data types and functions has not yet been forgotten during ongoing work on *EC*. Even if documentation comments have to be changed during subsequent work, this is justifiable compared to the absence of code documentation for too long.

After that, a reasonable action would be to extract all the user interface code from the *KnowledgeBase* class and place it separately, either as free functions or in a class. The same applies to the processing of the knowledge base. It should be carefully isolated and implemented in a reusable as well as exchangeable way, possibly using explicitly a *stack* for *IF rules* and questions.

Another step could be to choose more appropriate container templates and remove unnecessary restrictions, such as the maximum number of variables, rule lines, and questions. This would make it easier to manage the elements stored in the container and improve efficiency. This will probably also change the order in which *IF rules* are evaluated and questions are asked. The tests should therefore be flexible enough to deal with the consequences of these changes. Of course, this step can be done before or after any of the above steps, but it should not be done before adequate automated tests are available.

A next interesting step could be the introduction of numeric variables and comparisons in addition to textual variables. Another interesting extension in this context could be the introduction of keywords for comparators, e.g., smaller or larger, or user-defined predicates.

A drastic change would be to replace the current knowledge base processing with a compiler that translates a textual knowledge base into a programmatic source file that can be incorporated into a *C++* program and processed directly. This could be either a complete replacement or an additional tool. Incidentally, this could allow the creation of a direct interface between the processing of a knowledge base and the rest of a program.

Of course, it is also conceivable to implement *EC* using other programming paradigms. Perhaps this would lead to a more compact code.

Last but not least, it would also be an interesting task to create a graphical user interface for one of the previous visions of improved *EC*'s.

In practice, none of these steps would be taken without carefully considering whether the result would be useful, necessary, and desired by potential users, because each of these steps requires a significant investment of resources. But to

learn more about programming concepts and practices, performing each proposed step provides an exciting experience.

7. References

- Abelson, H., & Kong, S.-C. (Eds.). (2019). *Computational Thinking Education* (1st ed. 2019). Springer Singapore : Imprint: Springer. <https://doi.org/10.1007/978-981-13-6528-7>
- Arithmetic operators*—*Cppreference.com*. (n.d.). Retrieved March 6, 2021, from https://en.cppreference.com/w/cpp/language/operator_arithmetic
- ASCII. (2021). In *Wikipedia*. <https://en.wikipedia.org/w/index.php?title=ASCII&oldid=998308027>
- Assignment operators*—*Cppreference.com*. (n.d.). Retrieved March 6, 2021, from https://en.cppreference.com/w/cpp/language/operator_assignment
- Breymann, U. (2023). *C++ programmieren: C++ lernen - professionell anwenden - Lösungen nutzen* (7., überarbeitete Auflage). Hanser.
- C++ Core Guidelines*. (n.d.). Retrieved February 6, 2022, from <http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rio-endl>
- C++ Standard Library Active Issues List*. (n.d.). Retrieved February 4, 2022, from <http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-active.html#2703>
- C numeric limits interface*—*Cppreference.com*. (n.d.). Retrieved March 9, 2021, from <https://en.cppreference.com/w/cpp/types/climits>
- Catchorg/Catch2*. (2021). [C++]. Catch Org. <https://github.com/catchorg/Catch2> (Original work published 2010)
- ChatGPT*. (n.d.). Retrieved August 8, 2023, from <https://chat.openai.com>
- Cheng, E. (2017). *Beyond infinity: An expedition to the outer limits of mathematics*. Basic Books.
- Converting constructor*—*Cppreference.com*. (n.d.). Retrieved July 15, 2022, from https://en.cppreference.com/w/cpp/language/converting_constructor
- cplusplus.com*—*The C++ Resources Network*. (n.d.). Retrieved March 8, 2021, from <https://www.cplusplus.com/>
- Cppreference.com*. (n.d.). Retrieved March 8, 2021, from <https://en.cppreference.com/w/>
- Diehl, S., Hartel, P., & Sestoft, P. (2000). Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16(7), 739–751. [https://doi.org/10.1016/S0167-739X\(99\)00088-6](https://doi.org/10.1016/S0167-739X(99)00088-6)

- DOSBox, an x86 emulator with DOS.* (n.d.). Retrieved July 20, 2021, from <https://www.dosbox.com/>
- Expert system. (2021). In *Wikipedia*. [https://en.wikipedia.org/w/index.php?title=Expert system&oldid=1053923528](https://en.wikipedia.org/w/index.php?title=Expert%20system&oldid=1053923528)
- Floating point literal—Cplusplus.com.* (n.d.). Retrieved March 6, 2021, from https://en.cppreference.com/w/cpp/language/floating_literal
- Flowchart. (2021). In *Wikipedia*. <https://en.wikipedia.org/w/index.php?title=Flowchart&oldid=1056979544>
- Fundamental types—Cplusplus.com.* (n.d.). Retrieved March 6, 2021, from <https://en.cppreference.com/w/cpp/language/types>
- GNU Make Manual—GNU Project—Free Software Foundation.* (n.d.). Retrieved June 8, 2021, from <https://www.gnu.org/software/make/manual/>
- Google/googletest.* (2021). [C++]. Google. <https://github.com/google/googletest> (Original work published 2015)
- GotW #9: Memory Management—Part I.* (n.d.). Retrieved March 16, 2021, from <http://www.gotw.ca/gotw/009.htm>
- Gregoire, M. (2020). *Professional C++* (5th ed.). John Wiley and Sons.
- Hecker, B. (2009, October 27). *Production or Expert Systems 1 Weaknesses of Expert.* <https://slidetodoc.com/production-or-expert-systems-1-weaknesses-of-expert/>
- Hungarian notation. (2021). In *Wikipedia*. [https://en.wikipedia.org/w/index.php?title=Hungarian notation&oldid=997846419](https://en.wikipedia.org/w/index.php?title=Hungarian%20notation&oldid=997846419)
- Implicit conversions—Cplusplus.com.* (n.d.). Retrieved March 6, 2021, from https://en.cppreference.com/w/cpp/language/implicit_conversion
- INFINITY - cplusplus.com.* (n.d.). Retrieved March 7, 2021, from <https://en.cppreference.com/w/c/numeric/math/INFINITY>
- inline specifier—Cplusplus.com.* (n.d.). Retrieved November 24, 2021, from <https://en.cppreference.com/w/cpp/language/inline>
- Input/output manipulators—Cplusplus.com.* (n.d.). Retrieved March 6, 2021, from <https://en.cppreference.com/w/cpp/io/manip>
- ISO/IEC. (2020, December). *ISO/IEC 14882:2020.* ISO. <https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/07/93/79358.html>
- Josuttis, N. (2020). *C++ Move Semantics—The Complete Guide* (First Edition). Leanpub.

- Karpathy, A. (2017, November 11). Software 2.0. *Andrej Karpathy*. <https://medium.com/@karpathy/software-2-0-a64152b37c35>
- Lambert, C. (n.d.). *Makefile Tutorial by Example*. Makefile Tutorial. Retrieved June 25, 2021, from <https://makefiletutorial.com>
- List of types of numbers. (2021). In *Wikipedia*. [https://en.wikipedia.org/w/index.php?title=List of types of numbers&oldid=1008444145](https://en.wikipedia.org/w/index.php?title=List_of_types_of_numbers&oldid=1008444145)
- NAN - *cppreference.com*. (n.d.). Retrieved March 7, 2021, from <https://en.cppreference.com/w/cpp/numeric/math/NAN>
- OECD Glossary of Statistical Terms—Expert system Definition*. (n.d.). Retrieved December 20, 2021, from <https://stats.oecd.org/glossary/detail.asp?ID=3384>
- Order of evaluation—Cplusplus.com*. (n.d.). Retrieved June 26, 2021, from https://en.cppreference.com/w/cpp/language/eval_order
- Package: Areas/expert/systems/esie/*. (n.d.). Retrieved July 20, 2021, from <https://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/expert/systems/esie/0.html>
- Part IV. Boost Test Library: The Unit Test Framework*. (n.d.). Retrieved July 1, 2021, from https://www.boost.org/doc/libs/1_45_0/libs/test/doc/html/utf.html
- Permutation. (2021). In *Wikipedia*. <https://en.wikipedia.org/w/index.php?title=Permutation&oldid=1008522451>
- Rational number. (2021). In *Wikipedia*. [https://en.wikipedia.org/w/index.php?title=Rational number&oldid=1019402968](https://en.wikipedia.org/w/index.php?title=Rational_number&oldid=1019402968)
- Reference declaration—Cplusplus.com*. (n.d.). Retrieved March 16, 2021, from <https://en.cppreference.com/w/cpp/language/reference>
- Rule-based machine learning. (2021). In *Wikipedia*. [https://en.wikipedia.org/w/index.php?title=Rule-based machine learning&oldid=1033562331](https://en.wikipedia.org/w/index.php?title=Rule-based_machine_learning&oldid=1033562331)
- Sciences, N. I. of E. H. (1993). *Environmental Health Perspectives: EHP*. U.S. Department of Health, Education, and Welfare, Public Health Service, National Institutes of Health, National Institute of Environmental Health Sciences. <https://books.google.de/books?id=S9lfaUqCLLYC>
- Sign function. (2021). In *Wikipedia*. [https://en.wikipedia.org/w/index.php?title=Sign function&oldid=1008894481](https://en.wikipedia.org/w/index.php?title=Sign_function&oldid=1008894481)
- sizeof operator—Cplusplus.com*. (n.d.). Retrieved March 9, 2021, from <https://en.cppreference.com/w/cpp/language/sizeof>

- Song, T. (n.d.). *Working Draft, Standard for Programming Language C++*. Retrieved March 8, 2021, from <https://eel.is/c++draft/>
- std::isinf*—*Cppreference.com*. (n.d.). Retrieved March 15, 2021, from <https://en.cppreference.com/w/cpp/numeric/math/isinf>
- std::isnan*—*Cppreference.com*. (n.d.). Retrieved March 15, 2021, from <https://en.cppreference.com/w/cpp/numeric/math/isnan>
- std::numeric_limits*—*Cppreference.com*. (n.d.). Retrieved March 10, 2021, from https://en.cppreference.com/w/cpp/types/numeric_limits
- std::size_t*—*Cppreference.com*. (n.d.). Retrieved March 9, 2021, from https://en.cppreference.com/w/cpp/types/size_t
- The C++ Standard Template Library (STL). (2015, December 7). *GeeksforGeeks*. <https://www.geeksforgeeks.org/the-c-standard-template-library-stl/>
- The Incredible Const Reference That Isn't Const. (2018, July 13). *Fluent C++*. <https://www.fluentcpp.com/2018/07/13/the-incredible-const-reference-that-isnt-const/>
- The Programmer's Corner «ESIE.ZIP» Miscellaneous Language Source Code*. (n.d.). Retrieved July 20, 2021, from <https://www.pcorner.com/list/MISC/ESIE.ZIP/INFO/>
- Turing, A. M. (1937). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1), 230–265. <https://doi.org/10.1112/plms/s2-42.1.230>
- Value categories*—*Cppreference.com*. (n.d.). Retrieved March 16, 2021, from https://en.cppreference.com/w/cpp/language/value_category
- Working Draft, Standard for Programming Language C++*. (n.d.). Retrieved March 8, 2021, from <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4713.pdf>
- Yohe, W. P. (1987). *Software Reviews: Expert System Inference Engine (Esie), Version 2.0* Publisher: Lighthouse, Inc., PO Box 16858, Tampa, FL 33617 Distributor: NCSU Software, Box 8101, Raleigh, NC 27695 (telephone: 919-737-3067) Year of Publication: 1986 Materials: One DS disk (documentation in disk files) Price: \$10 (\$145 for PC-WRITE , updates, printed documentation, and telephone support) Availability: IBM and compatibles System Requirements: 256K, one DS drive, DOS 2.0 or later Effectiveness: Good User-Friendliness: Good. *Social Science Microcomputer Review*, 5(3), 404–408. <https://doi.org/10.1177/089443938700500324>