INFORMATIK
für Mathematiker und Physiker

Eine Einführung in C++

Skript zur Vorlesung 252-0847-00
Herbstsemester 2012, ETH Zürich

Bernd Gärtner        Michael Hoffmann

1This book originates from a set of lecture notes written by Joachim Giesen and the authors in 2005.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>9</td>
</tr>
<tr>
<td>1.1 Why learn programming?</td>
<td>10</td>
</tr>
<tr>
<td>1.2 How to run a program</td>
<td>15</td>
</tr>
<tr>
<td>1.2.1 Editor</td>
<td>15</td>
</tr>
<tr>
<td>1.2.2 Compiler</td>
<td>15</td>
</tr>
<tr>
<td>1.2.3 Computer</td>
<td>17</td>
</tr>
<tr>
<td>1.2.4 Operating system</td>
<td>18</td>
</tr>
<tr>
<td>1.2.5 Platform</td>
<td>19</td>
</tr>
<tr>
<td>1.2.6 Details</td>
<td>19</td>
</tr>
<tr>
<td>2 Foundations</td>
<td>21</td>
</tr>
<tr>
<td>2.1 A first C++ program</td>
<td>22</td>
</tr>
<tr>
<td>2.1.1 Syntax and semantics</td>
<td>23</td>
</tr>
<tr>
<td>2.1.2 Comments and layout</td>
<td>25</td>
</tr>
<tr>
<td>2.1.3 Include directives</td>
<td>25</td>
</tr>
<tr>
<td>2.1.4 The main function</td>
<td>26</td>
</tr>
<tr>
<td>2.1.5 Values and effects</td>
<td>27</td>
</tr>
<tr>
<td>2.1.6 Types and functionality</td>
<td>27</td>
</tr>
<tr>
<td>2.1.7 Literals</td>
<td>27</td>
</tr>
<tr>
<td>2.1.8 Variables</td>
<td>27</td>
</tr>
<tr>
<td>2.1.9 Constants (I promised myself)</td>
<td>28</td>
</tr>
<tr>
<td>2.1.10 Identifiers and names</td>
<td>29</td>
</tr>
<tr>
<td>2.1.11 Objects</td>
<td>30</td>
</tr>
<tr>
<td>2.1.12 Expressions</td>
<td>30</td>
</tr>
<tr>
<td>2.1.13 lvalues and rvalues</td>
<td>31</td>
</tr>
<tr>
<td>2.1.14 Operators</td>
<td>31</td>
</tr>
<tr>
<td>2.1.15 Statements</td>
<td>35</td>
</tr>
<tr>
<td>2.1.16 The first program revisited</td>
<td>36</td>
</tr>
<tr>
<td>2.1.17 Details</td>
<td>38</td>
</tr>
<tr>
<td>2.1.18 Goals</td>
<td>40</td>
</tr>
<tr>
<td>2.1.19 Exercises</td>
<td>41</td>
</tr>
</tbody>
</table>
## CONTENTS

2.1.20 Challenges ................................................. 44

2.2 Integers ......................................................... 46
  2.2.1 Associativity and precedence of operators .......... 47
  2.2.2 Expression trees ........................................ 48
  2.2.3 Evaluating expressions ................................. 49
  2.2.4 Arithmetic operators on the type `int` ................ 50
  2.2.5 Value range ............................................ 54
  2.2.6 The type `unsigned int` ............................... 56
  2.2.7 Mixed expressions and conversions ................. 56
  2.2.8 Binary representation ................................ 57
  2.2.9 Integral types ......................................... 59
  2.2.10 Details ................................................. 60
  2.2.11 Goals .................................................. 64
  2.2.12 Exercises .............................................. 65
  2.2.13 Challenges ............................................ 67

2.3 Booleans ...................................................... 68
  2.3.1 Boolean functions ..................................... 68
  2.3.2 The type `bool` ....................................... 70
  2.3.3 Short circuit evaluation .............................. 73
  2.3.4 Details ................................................. 73
  2.3.5 Goals .................................................. 75
  2.3.6 Exercises .............................................. 75
  2.3.7 Challenges ............................................ 77

2.4 Control statements .......................................... 80
  2.4.1 Selection: if- and if-else statements ............. 80
  2.4.2 Iteration: for statements ............................ 81
  2.4.3 Blocks and scope ..................................... 85
  2.4.4 Iteration: while statements ......................... 90
  2.4.5 Iteration: do statements ............................. 92
  2.4.6 Jump statements ...................................... 93
  2.4.7 Equivalence of iteration statements .............. 95
  2.4.8 Choosing the “right” iteration statements ....... 97
  2.4.9 Details ................................................. 98
  2.4.10 Goals .................................................. 102
  2.4.11 Exercises .............................................. 103
  2.4.12 Challenges ............................................ 107

2.5 Floating point numbers ...................................... 109
  2.5.1 The types `float` and `double` ....................... 110
  2.5.2 Mixed expressions, conversions, and promotions ... 112
  2.5.3 Explicit conversions .................................. 113
  2.5.4 Value range ........................................... 114
  2.5.5 Floating point number systems ...................... 115
  2.5.6 The IEEE standard 754 ................................ 120
2.5.7 Computing with floating point numbers ........................................ 121
2.5.8 Details ........................................................................................... 125
2.5.9 Goals ............................................................................................. 127
2.5.10 Exercises ...................................................................................... 128
2.5.11 Challenges .................................................................................... 132

2.6 Arrays and pointers ............................................................................ 134
2.6.1 Array types ..................................................................................... 134
2.6.2 Initializing arrays ........................................................................... 136
2.6.3 Random access to elements ........................................................... 136
2.6.4 Arrays are primitive ........................................................................ 137
2.6.5 Iteration over a container ............................................................... 138
2.6.6 Pointer types and functionality ..................................................... 139
2.6.7 Array-to-pointer conversion .......................................................... 142
2.6.8 Pointers and const ......................................................................... 143
2.6.9 Pointer arithmetic .......................................................................... 144
2.6.10 Dynamic memory allocation ......................................................... 148
2.6.11 Arrays of characters ..................................................................... 151
2.6.12 Multidimensional arrays ............................................................... 155
2.6.13 Beyond arrays and pointers ......................................................... 166
2.6.14 Details ........................................................................................... 167
2.6.15 Goals ............................................................................................ 168
2.6.16 Exercises ...................................................................................... 169
2.6.17 Challenges .................................................................................... 173

3 Functions ............................................................................................... 177
3.1 A first C++ function ........................................................................... 178
3.1.1 Pre- and postconditions ................................................................. 179
3.1.2 Function definitions ....................................................................... 181
3.1.3 Function calls ................................................................................ 182
3.1.4 The type void ................................................................................ 183
3.1.5 Functions and scope ...................................................................... 184
3.1.6 Procedural programming .............................................................. 186
3.1.7 Arrays as function arguments ....................................................... 188
3.1.8 Modularization .............................................................................. 191
3.1.9 Using library functions .................................................................. 197
3.1.10 Details .......................................................................................... 199
3.1.11 Goals ............................................................................................ 200
3.1.12 Exercises ...................................................................................... 201
3.1.13 Challenges .................................................................................... 208

3.2 Recursion ............................................................................................ 212
3.2.1 A warm-up .................................................................................... 212
3.2.2 The call stack ................................................................................. 213
3.2.3 Basic practice ................................................................................ 214
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.12 Goals</td>
<td>298</td>
</tr>
<tr>
<td>4.3.13 Exercises</td>
<td>298</td>
</tr>
<tr>
<td>4.3.14 Challenges</td>
<td>300</td>
</tr>
<tr>
<td>A C++ Operators</td>
<td>303</td>
</tr>
<tr>
<td>Index</td>
<td>306</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction
1.1 Why learn programming?

You can tell I'm educated, I studied at the Sorbonne
Doctored in mathematics, I could have been a don
I can program a computer, choose the perfect time
If you've got the inclination, I have got the crime

Pet Shop Boys, Opportunities (1986)

This section explains what a computer program is, and why it is im-
portant for you not only to use computer programs, but also to write
them.

When people apply for a job these days, their resume typically contains a section
called computer skills. Items listed there might include Lotus Notes, Excel, or Photo-
Shop. These are the names of application programs, programs that have been written
by certain people (in the above cases, at Microsoft corporation) to be used by other
people (for example, a sales representative).

The computer skills section might also list items like HTML, Java, or C++. These
are the names of programming languages, languages used to instruct, or program, a
computer. Using a programming language, you can write the programs that will subse-
quently be used by others, or by yourself.

A computer program is a list of instructions to be automatically processed by a
computer. The computer itself is stupid—all the intelligence comes from the program.
In this sense, a program for the computer is like a cookbook recipe for someone who
cannot cook: even with very limited skills, impressive results can be obtained, through
a step-by-step instruction.

Most people simply use programs, just like they use cookbooks. A sales representa-
tive, for example, needs application programs as tools for his work. The fact that you
are reading this lets us believe that you potentially belong to the category of people who
also need to write programs.

There are many reasons for writing programs. Some employer might pay for it, some
bachelor course might require it, but ultimately, there is a deeper reason behind it that
we plan to explain next. The upshot is that nowadays, you cannot be a serious engineer,
let alone a serious scientist, without at least some basic programming skills. Even in less
serious contexts, we can recommend to learn programming, because it can bring about
a lot of fun and satisfaction.

In the twentieth century, computers have revolutionized the way science and engi-
neering are done. To be more concrete, we will underpin this with an example from
mathematics. You probably don't expect math to be mentioned first in connection with
computers; indeed, many mathematicians still use paper and pencil on a daily basis.
But what they write down has changed. Before computers were available, it was often
necessary to write down actual numbers, and to perform calculations with them by hand. This happened not so much in writing proofs for new theorems, but in the process of finding these theorems. This process often requires to go over many concrete examples, or counterexamples, in order to see certain patterns, or to discover that some statement is false. The computer has tremendously accelerated this process by taking over the routine work. When you look at a mathematician's notepad today, you still find greek letters and all kinds of strange symbols, but most likely no numbers larger than ten.

There is one topic that nicely illustrates the situation, and this is the search for Mersenne primes. In 1644, the French monk and mathematician Marin Mersenne established the following claim.

**Mersenne’s Conjecture.** The numbers of the form $2^n - 1$ are prime numbers for $n = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127, 257$, but for no other number $n < 257$.

Mersenne corresponded with many of the leading mathematicians at that time, so his conjecture became widely known. Up to $n = 7$, you can verify it while you read this, and in 1644, the conjecture was already verified up to $n = 19$.

It took more than hundred years until the next exponent on Mersenne’s list could be verified. In a letter to Bernoulli published in 1772, Leonhard Euler proved that $2^{31} - 1 = 2147483647$ is a prime number. But in 1876, another hundred years later, Mersenne posthumously received a heavy blow. Edouard Lucas proved that $2^{67} - 1 = 147573952589676412927$ is not a prime number (Lucas showed his passion for large numbers also when he invented the Tower of Hanoi puzzle). Lucas’s proof does not work the way you would expect: it does not exhibit a prime factor of $2^{67} - 1$ (the most direct way of proving that a number is not prime), but it uses a clever indirect argument invented by Lucas in the same year. The factorization of $2^{67} - 1$ remained unknown for another 25 years.

In 1903, Frank Nelson Cole was scheduled to give a lecture to the American Mathematical Society, whose title was ‘On the Factorization of Large Numbers’. Cole went to the blackboard, and without saying a single word, he first wrote down a calculation to obtain $2^{67} - 1$ by repeated multiplication with two. He finally had the number

$147573952589676412927$

on the blackboard. Then he wrote down another (much more interesting) calculation for the product of two numbers.

$761838257287 \times 193707721$

$\begin{array}{l}
761838257287 \\
6856544315583 \\
2285514771861 \\
5332867801009
\end{array}$
Cole had proved that $2^{67} - 1 = 761838257287 \cdot 193707721$, making the result of Lucas believable to everybody: $2^{67} - 1$ is not a prime number! He received standing ovations for this accomplishment and later admitted that he had worked on finding these factors every Sunday for the last three years.

Today, you can start a computer algebra program on your computer (a popular one is Maple), type in

```plaintext
ifactor(2^67-1);
```

and within less than a second get the output

$$(761838257287)(193707721)$$

To summarize: hundred years ago, a brilliant mathematician needed three years to come up with a result that much less brilliant people (we are not talking about you) could get in less than a second today, using a computer and the right program. This seems disturbing at first sight, and thinking about the precious time of his life Cole devoted to the problem, you may even feel sorry for him. You shouldn't; rather, the story has three important lessons in store.

**Tool skills.** Lesson one is that Cole's calculations were extremely difficult, given the tools he had (paper, pencil, and probably very good mental arithmetic). Given the tools you have (the computer and a computer algebra program called Maple), Cole's calculations are easy routine. We are sure that Cole would feel sorry for anyone using these new tools only to reproduce some hundred-year old calculation. Useful new tools lead to new possibilities and challenges. On the one hand, this allows you to do more than you could do before; on the other hand it also forces you to do more if you want to keep up with the developments. Whatever you do, nowadays you must acquire and maintain at least some basic knowledge of computers and application programs.

**Problem skills.** Lesson two is that tool skills alone would not have helped Cole to factor $2^{67} - 1$. Cole also was a good mathematician who knew a lot of theory he could use to save calculations. This is the reason why he "only" needed three years.

Even nowadays, computers and application programs are not everything. Factoring $2^{67} - 1$ is easy because this is a small number by today's standards. But factoring large numbers ($2^{1000}$ is considered large today; in a couple of years, it might be $2^{2000}$) is still a very difficult problem for which no efficient solutions are known. The problem
1.1. WHY LEARN PROGRAMMING?

of factoring large numbers is the most prominent problem for which most people must actually hope that no efficient solution will ever be found. The reason is that many cryptosystems that are in use (think of secure internet banking) are purely based on the practical impossibility of factoring large numbers. Therefore, the worst scenario would be that the “bad guys” discover first how to factor large numbers efficiently.

There are many other problems that are as far from a solution as they were in pre-computer days. Coming back to Mersenne, we still cannot characterize the exponents $n$ for which the number $2^n - 1$ is a prime number. We don’t even know whether there are infinitely many such Mersenne primes. If you plan to make a contribution here, you should not buy a faster computer with the latest version of Maple, but study math. Even in the case of problems for which computers can really contribute to (or actually find) the solution, you typically need to have a deep understanding of the problem in order to know how to use the computer. If you want to become an engineer or a scientist, you must acquire and maintain a profound knowledge about the problems you will be dealing with. This fact was true hundred years ago, and it is still true—computers have not yet learned to solve interesting problems by themselves.

**Programming Skills.** Lesson three is one that Cole did not live to see: nowadays, problem-specific knowledge can be turned into problem-specific computer programs. That way, the state of the art concerning Mersenne primes has advanced quite far. It turned out that Mersenne had made five mistakes: $n = 67$ and $n = 257$ in Mersenne’s list do not lead to prime numbers; on the other hand, Mersenne had “forgotten” the exponents $n = 61, 89$ and $107$.

As of September 2009, we know 47 Mersenne primes, the largest of which has an exponent of $n = 43,112,609$ (see the GIMPS project at [www.mersenne.org](http://www.mersenne.org)). But don’t believe that this one was found with off-the-shelf programs.

Problems occurring in the daily life of an engineer or a scientist are often not easy to solve, even with a computer and standard software at hand. In order to attack them, you need tool skills for the routine calculations, and problem skills to understand and extract the aspects of the problem that can in principal be solved by a computer. But in the end, you need programming skills to actually do it.

**The art of computer programming.** To conclude this section, let us be honest: for many people (including the authors of this book), the process of writing programs has some very non-utilitarian aspects as well. We have mentioned two of them before: fun and satisfaction. We could add mathematical beauty and ego boost. In one way or another, every passionate programmer feels at least a little bit like an artist.

The prime advocate of this view on programming is Donald E. Knuth. He is the author of a monumental and seminal series of seven books entitled *The Art of Computer Programming*. Starting with Volume I in 1968, three of the seven volumes are published by now. Drafts of Volume IV circulate since 2005, and the planned release date of Volume V is 2015 (it should be added that Knuth was born in 1938, and on his webpage [http://www.cs.tufts.edu/~cse65/](http://www.cs.tufts.edu/~cse65/)).
//www-cs-faculty.stanford.edu/~knuth/taocp.html, he at least implicitly mentions the possibility that Volumes VI and VII will not be written anymore).

Let Knuth have the final say here (a quote from the beginning of Volume I):

The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music.
1.2. HOW TO RUN A PROGRAM

1.2 How to run a program

In Paris they just simply opened their eyes and stared when we spoke to them in French! We never did succeed in making those idiots understand their own language.

Mark Twain, The Innocents Abroad (1869)

This section explains what it really means to “write a program”, and how you enable the computer to run it. For this, we describe the ingredients involved in the process: the editor, the compiler, the computer itself, and the operating system. Computer, compiler and operating system together form the platform on which you are writing programs.

1.2.1 Editor

Writing a program is not so different from writing a letter. One composes a text, that is, a (hopefully) meaningful sequence of characters. Usually, there are certain conventions on how such a text is structured, and the purpose of the text is to transport information.

What has been said so far applies to both letters and programs. But when writing a program, there is another aspect that has to be taken into account: A program has to be “read” by a computer, meaning that it must be available to the computer in electronic form. In the future, we might be able to orally dictate the program to the computer, but nowadays, the common way is to use a keyboard and simply type it in. An editor is an application program that allows you to display, modify, and electronically store such typed-in text. The use of editors is not restricted to programming, of course. With some still existing romantic exceptions, even letters are composed using editors such as Word.

1.2.2 Compiler

Making a program available to the computer in electronic form is usually not enough. The machine language a computer can understand directly is very primitive and quite different from natural languages.

Writing the programs in machine language is no viable alternative, since that would require to break the program into a large number of primitive instructions that the computer can understand. This is like telling your friend to come over for dinner by telling her which muscles to move in order to get to your place.

Moreover, machine languages vary considerably between different computers. That is, in order to use a program written for one specific computer A on a different computer B, one first has to translate the program from the machine language of A to the
CHAPTER 1. INTRODUCTION

machine language of B. This process, called porting, can be very cumbersome if the machine languages of A and B are substantially different. Also, porting can only be done with a detailed knowledge of the peculiarities of the involved computers. But this type of knowledge is not generally worthwhile to acquire, as it is tied to one very specific computer. As soon as this computer is replaced by another one, major parts of such computer-specific knowledge become worthless and have to be rebuilt from scratch.

To reduce this undesirable entanglement of computers and programs, and to allow us to write programs in less primitive language, (high-level) programming languages have been developed. These are standardized languages that form a kind of compromise between natural languages and machine language. Indeed, the use of the word “compromise” is justified because there are two conflicting goals: On the one hand, we would like to write programs in a language that is as close to natural language as possible. On the other hand, we have to make the computers understand the programming language as well; this task is obviously much easier if the programming language is close to machine language.

What does it mean “to make the computers understand the programming language”? In the end, any program has to be translated into machine language. The process of this translation is called compilation. Now you will probably ask: “Where is the benefit of this whole programming language concept? In order to do the translation I still have to know all these computer-specific details, don’t I?” Right. If you would have to translate the program yourself. The key is: You are not supposed to translate it yourself. Instead, let a program do it for you. Such a program is referred to as a compiler; it translates a given program in a programming language, the source code, into a program in machine language, the executable. See Figure 1 for an illustration. The picture of the executable is somewhat inappropriate, since it does not show what the computer gets to see after compilation, but rather what you might see when you (accidentally) load the executable into the editor. The main point that we are trying to make here is that the executable is not human-readable.
In summary: The big benefit of (high-level) programming languages is that they abstract from the capabilities of specific computers. Programs written in a high-level language can be run on all kinds of computers, as long as a compiler for the language is available on the particular computer.

1.2.3 Computer

If you are not interested in writing compilers, it is not necessary to understand in detail how a computer works. But there are some basic principles behind the design of most computers that are important to understand. These principles form the von Neumann architecture, and they are important, since almost all programming languages are tailored to the von Neumann architecture.

Every computer with von Neumann architecture has a random access memory (RAM, or simply main memory), and a central processing unit (CPU, or simply processor). The main memory stores the program to be run, but also data that the program requires as input, and data that the program produces as output. The processor is the “brain” of the computer: it executes the program, meaning that it carries out the sequence of instructions prescribed by the program in machine language.

Main memory. You can think of the computer’s main memory as a long row of switches, each of them being either on or off. During program execution, switches are flipped. At any time, the memory content—the current positions of all switches—defines the program state. The program state completely determines what happens next. Conceptually, we also consider user input and program output as part of the program state, even though the corresponding “switches” might be in the user’s brain, or on printed paper.

Since modern computers are capable of flipping several switches at the same time, consecutive switches are grouped into memory cells. The positions of all switches in the cell define the content of the cell; in more abstract terms, the switches are called bits, each of them capable of storing one of the numbers \(0, 1\). The memory cells are usually called bytes and represent the smallest segments of memory that can individually be manipulated. In this sense, you can interpret the content of a memory cell as a binary number with, for example, 8 digits.

The computer may be able to manipulate more than 8 bits simultaneously; a typical value is 32 bits, or 4 bytes. We also say that we have a 32-bit machine, or a 32-bit system.

Each memory cell is uniquely identified by its address. You can think of the address simply as the position of the memory cell in the list of all memory cells.

To look up bit values, or to flip bits within a specific memory cell, the cell has to be accessed through its address. Think of a robot arm with 8 fingers that you can tell to move to memory cell number 17.

The term random access refers to a physical property of the computer’s memory:
the time it takes to access a cell (to "move to its bits") is the same for all cells; in particular, it does not depend on the address of the cell. When you think in terms of the robot arm analogy, it becomes clear that random access cannot be taken for granted. It is not necessary to discuss the physical means by which random access is realized; the important point here is that random access frees us from thinking about where to store a data item in order to access it efficiently.

Processor. You can think of the computer's processor as a box that is able to load and then execute the machine language instructions of a program in order. The processor has some memory cells of its own, called registers, and it can transfer data from the computer's main memory to its registers, and vice versa. The register contents are also part of the program state. Most importantly, the processor can perform a fixed set of simple operations (like adding or subtracting register contents), directly corresponding to the machine language instructions. This is where the functionality of the whole program comes from in the end. Even very complicated and useful programs can be put together from a simple set of machine language instructions.

A single instruction acts like a mathematical function: given the current program state, a valid instruction generates a new and well-defined next program state. This implies that every sequence of instructions, and in particular the whole program has a determined behavior, depending on the initial program state.

1.2.4 Operating system

We have seen that in order to write a program and run it, you first have to start an editor, type in the program, then call the compiler to translate the program into machine language, and finally tell the computer to execute it. In all this "starting", "calling" and "telling", you rely on the computer's operating system (OS), a program so basic that you may not even perceive it as a program. Popular operating systems are Windows, Unix, Linux, and Mac OS.

For example, you may start the editor by clicking on some icon, or by using a command shell (a text window for typing in commands; under Unix and Linux, this is still the default working mode). In both cases, the operating system makes sure that the editor program is loaded into the main memory, and that the processor starts executing it. Similarly, when you store your written program, the operating system allocates space for it on the hard disk and associates it with the file name you have provided.

A computer without operating system is like a car without tires, and most computers you can buy come with a pre-installed operating system. It is important to understand, though, that the operating system is not inextricably tied to the computer: you can take your "Windows PC" and reinstall it under Linux.
1.2. HOW TO RUN A PROGRAM

1.2.5 Platform

The computer, its operating system and the compiler are together referred to as the platform on which you are writing your programs. The editor is not part of the platform, since it does not influence the behavior of the program.

In an ideal world, there is no need for you to know the platform when you are writing programs in a high-level programming language. Recall that the plan is to delegate the platform-specific aspects to the compiler. A typical such platform-specific aspect is the number of bits that can be manipulated together. This is mostly 32 these days, but for some computers it is 64, and for very primitive computers (like they are used in smart cards, say), it can be less than 32.

When you are using or relying on machine-oriented features of the programming language, platform-specific behavior might be the result. Many high-level programming languages have such low-level features to facilitate the translation into efficient machine language.

Your goal should always be to write platform-independent code, since otherwise, it may be very difficult to get your program to run on another computer, even if you have a compiler for that computer. This implies that certain features should be avoided, even though it might seem advantageous to use them on a specific platform.

1.2.6 Details

Von Neumann's idea of a common memory for the program and the data seems obvious from today's point of view, but the earliest computers like Konrad Zuse's Z3 didn't work that way. In the Z3, for example, the memory for the program was a punch tape, decoupled from the input and output device, and from the main memory.

An interesting feature of the von Neumann architecture is that it allows self-modifying programs. These are popular among the designers of computer viruses, for example.

The von Neumann architecture with its two levels of memory (main memory and processor registers) is an idealized model, and we are implicitly working under this model throughout the course.

The reality looks more complicated. Modern computers also have a cache, logically belonging to the main memory, but allowing much faster access to memory cells (at the price of a more elaborate and expensive design). The idea is that frequently needed data are stored in the cache to speed up the program.

While caching is certainly a good thing, it makes the life of a programmer more difficult: you can no longer rely on the fact that access time to data is independent from where they are stored. In fact, to get the full performance benefit that caching can offer, the programmer has to make sure that data are accessed in a cache-coherent way. Doing this, however, requires some computer-specific knowledge about the cache, knowledge we were originally trying to avoid by using high-level programming languages. Luckily, we can often ignore this issue and (successfully) rely on the automatic cache management being offered. There is also a theoretical model for so-called cache-oblivious algorithms,
in which the algorithm does not know the parameters of the cache. Algorithms which are efficient under this model, are (in a certain sense) efficient for every concrete cache size.

In real-life applications, we also observe the phenomenon that the data to be processed are too large to fit into the computer's main memory. Operating systems can automatically deal with this by logically extending the main memory to the hard disk. However, the swapping that takes place when hard disk data to be accessed are transferred to the main memory incurs a severe performance penalty, much worse than poor cache usage. In this situation, it is often useless to rely on the automatic mechanisms provided by the operating systems, and the programmer is challenged to come up with input/output efficient programs.

Even when we extend the von Neumann architecture to include several layers of memory, there are computers that don't fit in. Most notably, there are parallel computers with more than one processor. In fact, even consumer PCs have more than one processor these days. Writing efficient programs for such a computer is a task entirely different from programming for the von Neumann architecture. To take full advantage of the parallelism, programs have to be decomposed into independent parts, each of which is then run by one of the processors. In many cases, this is not at all a straightforward task, and specialized programming languages have to be used.

A recent successful alternative to parallel computers are networks of single-processor computers. You can even call this a computer architecture. Finally, there are quantum computers that are based on completely different physical principles than the von Neumann architecture. "Real" quantum computers cannot be built yet, but as a theoretical model, quantum computers exist, and algorithms are already being developed in this promising model of computation.
Chapter 2

Foundations
2.1 A first C++ program

The basic tool for the manipulation of reality is the manipulation of words. If you can control the meaning of words, you can control the people who must use the words.

Philip K. Dick, How to Build a Universe That Doesn’t Fall Apart Two Days Later (1978)

This section presents a first complete C++ program and introduces the syntactical and semantical terms necessary to understand all its parts.

Here is our first C++ program. It asks for a number $a$ as input and outputs its eighth power $a^8$. If you have never seen a C++ program before, even this short one might look scary, since it contains a lot of strange-looking symbols and words that are not found in natural language. On the other hand, this is good news: as short as it is, this program already contains many important features of the C++ language. Once we have gone through them in this section, this program (and even other, bigger programs) won’t look scary anymore.

```cpp
// Program: power8.cpp
// Raise a number to the eighth power.

#include <iostream>

int main()
{
    // input
    std::cout << "Compute a^8 for a =? ";
    int a;
    std::cin >> a;

    // computation
    int b = a * a; // b = a^2
    b = b * b; // b = a^4

    // output b * b, i.e., a^8
    std::cout << a << "^8 = " << b * b << "\n";
    return 0;
}
```

Program 1: `../progs/lecture/power8.cpp`
2.1. A FIRST C++ PROGRAM

If you compile this program on your computer and then run the executable file produced by the compiler, you find the following line on the standard output. Typically, the standard output is attached to some window on your computer screen.

Compute \(a^8\) for \(a = \)?

You can now enter an integer, e.g. 2, using the keyboard. After pressing ENTER, the output on your screen reads as follows.

Compute \(a^8\) for \(a = \) 2
\[2^8 = 256.\]

Before discussing the program `power8.cpp` in detail, let us go over it once quickly. The lines starting with two slashes `//` are comments; they document the program such that it can easily be understood by a (human) reader. Line 4 contains an `include-directive`; in this case, it indicates that the program uses the input/output library `iostream`. The `main function` which is the heart of every C++ program spans lines 6–20. This function is called by the operating system when the program is started; it ends with a `return statement` in line 19. The value 0 is returned to the operating system, which by convention signals that the program terminated successfully.

The main function is divided into three parts. First, in lines 8–11 the input number is read. Line 9 outputs a message to the user that tells her which kind of input the program expects. In line 10 a `variable` `a` is declared that acts as a placeholder to store the input number. The keyword `int` indicates that `a` is an integer. In line 11, finally, the variable `a` receives its value from the input.

Then in lines 13–15 the actual computation takes place. In line 14, a new variable `b` is declared which acts as a placeholder to store the result of the computation. The variable `b` is initialized to the product `a * a`. Line 15 computes the product `b * b`, that is, \(a^4\) and stores this result again in `b`.

The third part in lines 17–18 provides the program output. Part of it is the computation of the product `b * b`, that is, \(a^8\).

2.1.1 Syntax and semantics.

In order to understand the program `power8.cpp` in detail, and more importantly, to write programs yourself later, you need to know the rules according to which programs are written. These rules form the `syntax` of C++. You further need to know how to interpret a program ("what does the program do?") and this is determined by the `semantics` of C++. Even a program that is well-formed according to the C++ syntax may be `invalid` from a semantical point of view. A `valid` program is one that is syntactically and semantically correct.

It's the same with natural language: grammar tells you what sentences are, but the interpretation of a sentence (in particular, whether it makes sense at all) requires a concept of meaning.

When a program is invalid, the compiler may output an error message, and this will definitely happen when the program contains `syntax errors`, violations of the syntactical
rules. A program that is semantically invalid may compile without errors, but we are not allowed to make any assumptions about its behavior; the program could run fine, for example if the semantical error in question has no consequences on a particular platform. On other platforms, the program may behave strangely, or crash. Even on the same platform, it might work sometimes, but fail at other times. We say that the program’s behavior is undefined. Clearly, one should avoid writing programs that exhibit undefined behavior.

The syntax of C++ is specified formally in a mathematical language. The description of the semantics is less strict; it rather resembles the text of a law, and as such it suffers from omissions and possible misinterpretations. The official law of C++ covering both syntax and semantics, is the ISO/IEC standard 14882 from 2011. \[1\]

While such a formal specification is indispensible (otherwise, how should a compiler know whether your program text is actually a C++ program, and what it is supposed to do?), it is not suitable for learning C++. Throughout this book, we explain the relevant syntactical and semantical terms in natural language and by example. For the sake of readability, we will often not strictly distinguish between syntactical and semantical terms: some terms are most naturally introduced as having both syntactical and semantical aspects, and it depends on the context which aspect is relevant.

Unspecified and implementation defined behavior. Sometimes, even valid programs behave differently on different platforms; this is one of the more ugly aspects of C++ that we’d prefer to sweep under the rug. Unfortunately, we can’t ignore the issue completely, since it occasionally pops up in “real life”.

There are two kinds of platform-dependent behavior. The nicer one is called implementation defined behavior.

Whenever the C++ standard calls some aspect of the language “implementation defined”, you can expect your platform to contain documentation that fully specifies the aspect. The typical example for such an implementation defined aspect is the number of bits that can be manipulated at once, see Section 1.2.3. In case of implementation defined aspects and resulting behavior, the C++ standard and the platform together completely determine the actual behavior.

The less nice kind is called unspecified behavior, coming from some unspecified aspect of the language. Here you can rely on a well-defined and usually small set of possible specifications, but the platform is not required to contain a full specification of the aspect. A typical example for such an unspecified aspect is the evaluation order of operands within an expression, see Section 2.1.12.

In writing programs, unspecified aspects cannot always be avoided, but usually, some care ensures that no unspecified or even undefined behavior results.

\[1\] This book is based on the previous version of the document from 1998, and as far as the book content is concerned, the differences between the previous and the new version are minor. A future version of this book will also address some of the new features of C++ that are officially available since 2011.
2.1.2 Comments and layout

Every good program contains comments, for example

```cpp
// Program: power8.cpp
// Raise a number to the eighth power.
```

A comment starts with two slashes // and continues until the end of the line. Comments do not provide any functionality, meaning that the program would do exactly the same without them. Why is a program without comments bad, then? We do not only write programs for the compiler to translate them into executables, but we also write them for other people (including ourselves) to read, modify, correct or extend them.

Without comments, the latter tasks become very tedious when the program is not completely trivial. Trust us: Even you will not be able to understand your own programs after a couple of weeks, without comments. There is no standard way of writing comments, but we will follow some common-sense guidelines. One of them is that every program—even if it is very simple—should start with one or more lines of comments that mention the program’s name and say what it does. In our case, the above two lines fully suffice.

Another key feature of a readable program is its layout; consider the version of power8.cpp shown in Program 2. We have removed comments, and all “unnecessary” layout elements like spaces, line breaks, blank lines, and indentations.

```cpp
#include <iostream>
int main(){std::cout <<"Compute a^8 for a =? ";
int a;std::cin >>a;int b=a*a;b=b*b;std::cout <<
a<<"^8 = "<<b*b<<"\n";return 0;}
```

Program 2: ../progs/lecture/power8_consolidated.cpp

The compiler is completely ignorant about these changes, but a person reading the program will find this condensed version quite difficult to understand. The purpose of a good layout is to visualize the program structure. This for example means that logical blocks of the program should be separated by blank lines, or that one line of source code should be responsible for only one thing. Indentation, like power8.cpp has it between the pair of curly braces, is another indispensable ingredient of good layout, although you will only later be able to fully appreciate this.

Typically, collaborative software projects have layout guidelines, making sure that everybody in the project can easily read everybody else’s code. At the level of the simple programs discussed in this book, such formal guidelines are not necessary; we simply adhere to standard guidelines that have proven to work well in practice, and that are being used in almost every other book on C++ as well.

2.1.3 Include directives

Every useful program contains one or more include directives, such as
#include <iostream>

Usually, these appear at the very beginning of the program. #include directives are needed since, in C++, many important features are not part of the core language. Instead, they are implemented in the so-called standard library which is part of every C++ implementation. A library is a logical unit used to group certain functionality and to provide it to the user in a succinct form. In fact, the standard library consists of several libraries one of which is the input/output library.

A library presents its functionality to the user in the form of one or several headers. Each such header contains information that is needed by the compiler. In order to use a certain feature from a library, one has to include the corresponding header into the program by means of an #include directive. In power8.cpp, we want to use input and output which are (maybe surprisingly) not part of the core language. The corresponding header of the standard library is called iostream.

A well-designed C++ library puts its functionality into a namespace. The namespace of the standard library is called std. Then, in order to access a feature from the library, we have to qualify its name with the namespace, like in std::cin (this is the feature that allows us to read input from the keyboard). This mechanism helps to avoid name clashes in which different features accidentally get the same name. At the same time, explicit qualification increases the readability of a program, as it is immediately apparent from which library a given feature comes. A name that is not qualified is called unqualified and usually corresponds to a feature defined in our own program.

2.1.4 The main function

Every C++ program must have a main function. The shortest program reads as follows.

```cpp
int main() { return 0; }
```

This program does nothing. The main function is called by the operating system when you tell it to run the program; but why is it a function, and what is "return 0;" supposed to mean? Just like a mathematical function, the main function can have arguments given to it upon execution of the program, and the computations within the curly braces yield a function value that is given back (or returned) to the operating system. In our case, we have written a main function that does not expect any arguments (this is indicated by the empty brackets () behind main) and whose return value is the integer 0. The fact that the return value must be an integer is indicated by the word int before main. By convention, return 0 tells the operating system that the program has run successfully (or that we don't care whether it has), while any other value explicitly signals failure.

In a strict mathematical sense, the main function of power8.cpp is utterly boring. The whole functionality of the program comes from the effect of the function. This effect is to read a number from the standard input and write its eighth power to the standard output. The fact that functions can have effects sets C++ apart from many functional programming languages.
2.1.5 Values and effects

The value and effect of a function are determined by the C++ semantics. Merely knowing the syntactical rules of writing functions does not tell us anything about values and effects. In this sense, value and effect are purely semantical terms.

For example, we have to know that in C++, the character 0 is interpreted as the integer 0 (although this is not difficult to guess). It is also important to understand that value and effect depend on the concrete program state in which the function is called.

2.1.6 Types and functionality

The word int is the name of a C++ type. This type is used since the program power8.cpp deals with integers. In mathematics, integers are modeled by the ring \((\mathbb{Z}, +, \cdot)\). This algebraic structure defines the integers in terms of their value range (the set \(\mathbb{Z}\)), and in terms of their functionality (addition and multiplication). In C++, integers can be modeled by the type int. Like a "mathematical type", a C++ type has a name, a value range, and functionality, defining what we can do with it. When we refer to a type, we will do so by its name. Note that the name is a syntactical aspect of the type, while value range and functionality are of semantical nature.

Conveniently, C++ contains a number of fundamental types (sometimes called built-in types) for typical applications. The type int is one of them. The major difference to the "mathematical type" \((\mathbb{Z}, +, \cdot)\) is that int has a finite value range only.

2.1.7 Literals

A literal represents a constant value of some type. For example, in line 19 of the program power8.cpp, 0 is a literal of type int, representing the value 0. For each fundamental type, it is separately defined how its literals look like, and what their values are. A literal can be seen as the syntactical counterpart of a value: it makes the value "visible" in the program.

2.1.8 Variables

The line

```cpp
int a;
```

contains a declaration of a variable. A variable represents a not necessarily constant value of some type. The variable has a name, a type, a value, and an address (typically in the computer's main memory; you can think of the address simply as the position of the variable in the main memory). The purpose of the address is to store and look up the value under this address. The reason for calling such an entity a variable is that its value can be changed by modifying the memory content at the corresponding address. In contrast, the name and type remain fixed.
When we refer to a variable, we will do so by its name. The declaration \texttt{int a} defines a variable with the following characteristics.

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>value</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>int</td>
<td>undefined</td>
<td>chosen by compiler/OS</td>
</tr>
</tbody>
</table>

You might wonder why this is called a \textit{definition} of a, even though it does not define the value of a. But recall that this value depends on the program state, and that the definition fully specifies \textit{how} the value is obtained: look it up at a's address. Saying that a variable \textit{has} a value is therefore somewhat imprecise, but we'll stick to it, just like mathematicians talk about \textit{function value} when they actually mean the value obtained by evaluating the function with concrete arguments. We even go one step further with our sloppiness: if a has value 2, for example, we also say that "a is 2". This is the way that programmers usually talk about variables and their values. We will get to know mechanisms for assigning and changing values of variables in Section \ref{sec:assigning-changing-values}.

In C++, it is good general practice to define a variable immediately before it is used for the first time. This practice improves the readability of your programs.

### 2.1.9 Constants (I promised myself)

Imagine that you wake up on a Sunday morning with a raging hangover, and you promise yourself that you will never touch alcohol again. But if there is no one to check whether you keep that promise, chances are that next Sunday you wake up with the next hangover.

In the ethical language of C++, you can give promises that you must keep. Here is an example of the most basic kind of promise. In writing

\begin{verbatim}
const int speed_of_light = 299792458;
\end{verbatim}

you define a variable \texttt{speed\_of\_light} whose value is 299,792,458 (m/s). The keyword \texttt{const} in front of the declaration is the promise that this variable will never change its value throughout the program. Such a variable is called a \textit{constant}. The type of \texttt{speed\_of\_light} is \texttt{const int}, the \textit{const-qualified} version of \texttt{int}.

The compiler will check whether you as the programmer keep your promise. Any subsequent lines of code that attempt to store a value under the address of \texttt{speed\_of\_light} will generate an error message. For example, if your program contains the two lines

\begin{verbatim}
const int speed_of_light = 299792458;
...
speed_of_light = 300000000;  // let's make it a bit easier
\end{verbatim}

the compiler will remind you of your promise not to change the value of \texttt{speed\_of\_light}. The \texttt{const} promise implies that declarations as in

\begin{verbatim}
const int speed_of_light;
\end{verbatim}

make no sense and are actually forbidden in the main program, as they would yield constants of eternally undefined value.
You don’t have to give a `const` promise, and since it seems like extra effort at first sight, why should you do it? Because it helps you (and others that may further develop your code in the future) to avoid programming errors. If you define a variable (such as `speed_of_light`) with the intention of keeping its value fixed once and for all, then you should use `const` to tell the compiler about your intention. If you don’t, it may (accidentally) happen that a ‘constant’ (such as the speed of light, or the VAT rate) gets modified; the resulting erroneous behavior of the program may be very hard to discover and to track down.

The whole point of high-level programming languages is to make the programmer’s life easier; the compiler is our friend and can help us to avoid many time-consuming errors. The `const` mechanism is like a check digit: by providing additional redundant data (the `const` keyword), we make sure that inconsistencies in the whole data set (the program) are automatically detected.

Also, consistently using `const` makes the program more readable, an aspect whose importance we have already stressed in Section 2.1.2. In reading

```cpp
const int speed_of_light = 299792458;
```

you immediately know that the program is working with a fixed `speed_of_light` throughout. We therefore advocate the following

**Const Guideline:** Whenever you define a variable in a program, think about whether its value is supposed to change or not. In the latter case, use the `const` keyword to turn the variable into a constant.

In existing code (of “real programmers”), this guideline is often not followed, and in the context of very short programs, it may even be perceived as pedantic. The authors prefer being pedantic to risking unnecessary programming errors. A program that follows the Const Guideline is called *const-correct*.

### 2.1.10 Identifiers and names

The name of a variable must be an *identifier*, according to the following definition, and it must be different from certain *reserved* names like `int`.

**Definition 1** An identifier is a sequence of characters composed of the 52 letters a…z and A…Z, the 10 digits 0…9, and the underscore (_). The first character has to be a letter.

A C++ program may also contain other names, for example the *qualified* names `std::cin` and `std::cout`. The C++ syntax specifies what a name is, while the C++ semantics tells us what the respective name refers to in a given context.
2.1.11 Objects

An object is a part of the computer's main memory that is used by the program to store a value. An object has an address, a type, and a value of its type (determined by the memory content at the object's address).

With this definition, a variable can be considered as a named object, but we may also have unnamed objects. Although we can't show an example for an unnamed object at this point, we can argue that unnamed objects are important.

In fact, if you want to write interesting programs, it is absolutely necessary to work with objects that are not named by variables. This can be seen by the following simple thought experiment: suppose that you have written a program that stores a sequence of integers to be read from a file (for example, to sort them afterwards). Now you look at your program and count the number of variables that it contains. Say this number is 31. But in these 31 variables, you can store no more than 31 integers. If your program is of any practical use, it can certainly store a sequence of 32 integers, but then there must be at least one integer that cannot be stored under a variable name.

2.1.12 Expressions

In the program power8.cpp, three character sequences stand out, because they look familiar and are chiefly responsible for the functionality of the program: these are the character sequences $a \times a$ in line 14 and $b \times b$ in lines 15 and 18.

An expression represents a computation involving other expressions. More precisely, an expression is either a primary expression, for example a literal or a name, or it is a composite expression. A composite expression is obtained by combining expressions through certain operations, or by putting a pair of parentheses () around an expression.

The expression $a \times a$ is an arithmetic expression, involving numeric variables (actually, the names of the variables, but for the sake of readability, we suppress this subtlety) and the multiplication operator, just like we know it from mathematics. According to our above definition, $a \times a$ is a composite expression, built from the multiplication operator and the two primary expressions $a$ and $a$.

According to the above definition, an expression is a syntactical entity, but it has semantical aspects as well: every expression has a type, a value of this type, and possibly an effect. The type is fixed, but the value and the effect only materialize when the expression gets evaluated, meaning that the computation it represents is carried out. Evaluating an expression is the most frequent activity going on while a C++ program is executed; the evaluation computes the value of the expression and carries out its effect (if any).

Type and value of a primary expression are determined by its defining literal, or by type and value of the entity behind its defining name. Primary expressions have no effect. Type, value and effect of a composite expression are determined by the involved operation, depending on the values and effects of the involved sub-expressions. Putting parentheses () around an expression yields an expression with the same type, value and
2.1. A FIRST C++ PROGRAM

effect.

The expression \( a \times a \), for example, is of type \( \text{int} \), and not unexpectedly, its value is the square of the value of \( a \). The expression has no effect. The expression \( b = b \times b \), built from the assignment operator and the two expressions \( b \) and \( b \times b \), has the same type and value as \( b \times b \), but it has an additional effect: it assigns the square of \( b \) back to \( b \). (This is a shorthand for the correct, but somewhat clumsy formulation that the new value of \( b \) is set to the square of the old value of \( b \).)

We say that an expression is *evaluated* rather than *executed*, because many expressions do not have an effect, so that their functionality is associated with the value only. Even for expressions with effect, some books use the term *side effect* to emphasize that the important thing is the value. The C++ entities chiefly responsible for effects are the *statements* to which we get below.

We want to remark that the *only* way of accessing an expression's value is to evaluate it, and this also carries out its effect. You cannot get the value without the effect.

### 2.1.13 Lvalues and rvalues

An lvalue is an expression that has an address. In the program `power8.cpp`, the variable \( b \) is an lvalue, and its address is the address of the variable \( b \).

The value of an lvalue is defined as the value of the object at its address. An lvalue can therefore be viewed as the syntactical counterpart of an object: it gives the object a (temporary) name and makes it “visible” within a C++ program. We also say that the lvalue refers to the object at its address.

In particular, every variable is an lvalue. But lvalues provide a means for accessing and changing object values, even without having a corresponding variable. As we will see in Section 2.1.14 below, the expression `std::cout << a` in line 18 is such an lvalue.

Every expression that is not an lvalue is an rvalue. For example, literals are rvalues: there is no address associated with the `int`-literal `0`, say. Putting a pair of parenthesis around an lvalue yields an lvalue, and similarly for rvalues.

The terms lvalue and rvalue already indicate that we think about them not so much in terms of expressions, but rather in terms of their values. We will often identify an lvalue with the object it refers to, and an rvalue simply with its value.

### 2.1.14 Operators

Line 14 of `power8.cpp`, for example, features the binary multiplication operator \( \ast \).

Like a function, an operator expects arguments (here also called operands) of specified types, from which it computes a return value of a specified type, according to its functionality. In addition, these computations may have an effect.

This was the semantical view; on the syntactical level, the operands as well as the composite expression (built from the operator and its operands, see Section 2.1.12), are
expressions; the operator specifies for each of them whether it is an lvalue or an rvalue. If the composite expression is an lvalue, the operator is said to return the object referred to by the lvalue. If the composite expression is an rvalue, the operator simply returns its value.

The number of operands is called the *arity* of the operator. Most operators have arity 1 (unary operators) or 2 (binary operators).

Whenever an rvalue is expected as an operand, it is also possible to provide an lvalue. In this case, the lvalue will simply be interpreted as an rvalue, meaning that its address is only used to look up the value, but not to change it. This is known as *lvalue-to-rvalue conversion*. In stating that an operand must be an rvalue, the operator therefore guarantees that the operand's value remains unchanged; by expecting an lvalue, the operator explicitly signals its intention to change the value.

**Evaluation of composite expressions.** When a composite expression involving an operator gets evaluated, the operands are evaluated first (recall that this also carries out the effects of the operands, if any). Based on the resulting values, the operator computes the value of the composite expression. The latter computations may have additional effects, and all effects together form the effect of the composite expression.

The order in which the operands of a composite expression are evaluated is (with rare exceptions) unspecified, see also Section 2.1.1.

Therefore, if the effect of one operand influences values or effects of other operands, value and effect of the composite expression may depend on the evaluation order. The consequence is that value and effect of the composite expression may be unspecified as well.

Since the compiler is not required to issue a warning in such cases, it is the responsibility of the programmer to avoid any expression whose value or effect depends on the evaluation order of operands.

**Operator specifics.** What is it that sets operators apart from functions? On the one hand, there is only a finite number of possible operator *tokens* such as * or -. Many of these tokens directly correspond to well-known mathematical operator symbols indicating the functionality of the operator. Unfortunately, the token = corresponds to mathematical assignment :=, and not to mathematical equality =, a constant source of confusion for beginners.

On the other hand, and most conveniently, operator calls do not have to obey the usual function call notation, like in f(x, y). After all, we want to write a * a in a program, and not *(a, a). In summary, operators let us write more natural and more readable code.

Four different operators (all of them binary) occur in power8.cpp, namely the multiplication operator *, the assignment operator =, the input operator >>, and the output operator <<. Let us discuss them in turn.
2.1. A FIRST C++ PROGRAM

Multiplication operator. The multiplication operator \(*\) expects two rvalue operands of some type \(T\), and it returns the product of its two operands as an rvalue. The multiplication operator has no effect on its own.

Assignment operator. The assignment operator \(=\) expects an lvalue of some type \(T\) as its first operand, and an rvalue of the same type as its second operand. It assigns the value of the second operand to the first operand and returns the first operand as an lvalue. In our program power8.cpp, the expression \(b = b * b\) therefore sets the value of \(b\) to the square of its previous value, and then returns \(b\).

In fact, the letter "l" in the term lvalue stands for the fact that the expression may appear on the left hand side of an assignment. Similarly, the term rvalue signals an expression that may appear only on the right hand side of an assignment.

Input Operator. In power8.cpp, the composite expression \(\text{std::cin} \gg a\) in line 11 sets the variable \(a\) to the next value from the standard input, usually the keyboard.

In general, the input operator \(\gg\) expects as its first operand an lvalue referring to an input stream. The second operand is an lvalue of some type \(T\). The operator sets the second operand to the next value read from the input stream and returns the stream as an lvalue.

An input stream represents the state of some input device. We think of this device as producing a continuous stream of data that can be tapped to provide input on demand. Under this point of view, the state of the stream corresponds to the sequence of data not yet read. In setting the value of its second operand, the input operator removes one data item from the stream to reflect the fact that this item has now been read. For this, it is important that the stream comes as an lvalue. Conceptually, an input stream is also considered part of the program state.

How much of the data is read as one item, and how exactly it is interpreted as a value of type \(T\) highly depends on the type \(T\) of the second operand. For now, it is enough to know that this interpretation is readily defined for the type \(\text{int}\) and for the other fundamental types that we will encounter in the following sections.

In C++, the lvalue \(\text{std::cin}\) refers to the variable \(\text{cin}\) defined in the input/output library, and this variable corresponds to the standard input stream.

It is up to the program's caller to fill the standard input stream with data. For example, suppose that the program was started from a command shell. Then usually, while the program is running, all input to the command shell is forwarded to the program's standard input stream. It is also possible to redirect a program's standard input stream to read data from a file instead.

The fact that the input operator returns the input stream is not accidental, as it allows to build expressions involving chains of input operations, such as \(\text{std::cin} \gg x \gg y\). We will discuss this mechanism in detail for the output operator below.
Output Operator. In `power8.cpp`, the composite expression `std::cout << a` in line 18 writes the value of `a` to the *standard output*, usually the computer screen.

In general, the output operator `<<` expects as its first operand an lvalue referring to an *output stream*. The second operand is an rvalue of some type `T`. The operator writes the value of the second operand to the output stream and returns the output stream as an lvalue.

An output stream represents the state of some output device. We think of this device as storing the continuous stream of output data that is generated by the program. In writing to the stream, the output operator therefore changes the stream state, and this makes it necessary to provide the stream as an lvalue. Conceptually, an output stream is also considered part of the program state.

It depends on the type `T` in which format the second operand's value is written to the stream; for the type `int` and the other fundamental types, this format is readily defined.

C++ defines a standard output stream `std::cout` and a *standard error* stream `std::cerr` in the input/output library.

It is up to the program's caller to process these output streams. For example, suppose that the program was started from a command shell. Then usually, while the program is running, both standard output stream and standard error stream are forwarded to the command shell. But it is also possible to redirect one or both of these streams to write to a file instead. This can be useful to separate regular output (sent to `std::cout`) from error output (sent to `std::cerr`).

As indicated above for input streams, it is possible to output several values through one expression, as in

```cpp
std::cout << a << "^8 = " << b * b << ".\n"
```

Maybe this looks a bit strange, because there is more than one `<<` operator token and more than two operands; but in mathematics, we also write `a + b + c` as a shortcut for either `(a + b) + c` or `a + (b + c)`; because addition is associative, we don't even have to specify which variant we intend.

In C++, such shortcuts are also allowed in order to avoid cluttering up the code with parentheses. But C++ operators are in general not associative, so we have to know the 'logical parentheses' in order to understand the meaning of the shortcut.

The operators `>>` and `<<` are *left-associative*, meaning that the above expression is logically parenthesized as follows.

```cpp
(((std::cout << a) << "^8 = ") << b * b) << ".\n"
```

Recall that the innermost expression `std::cout << a` is an lvalue referring to the standard output stream. Hence, this expression serves as a legal first operand for the next outer composite expression `(std::cout << a) << "^8 = "` and so on. The full expression therefore outputs the values of all expressions occurring after some `<<`, from left to right. The rightmost of these expressions ends with `\n` that causes a line break.

**Warning:** In using such nested output expressions, it is very easy to make mistakes
based on false assumptions about the evaluation order. As an example, consider the expression

```cpp
std::cout << (a = 10) << "," << a
```

where the variable `a` initially has value 5, say. You might expect that this outputs `10, 10`. But it is equally conceivable that the output is `10, 5`. The latter happens if the right operand `a` of the outermost composite expression

```cpp
((std::cout << (a = 10)) << ",") << a
```

is evaluated first, and the former happens if the left operand (whose effect includes changing the value of `a` to 10) is evaluated first. On the platform of the authors, the output is `10, 5`.

### 2.1.15 Statements

A statement is a basic building block of a C++ program, and it usually has an effect. The effect depends on the program state and materializes when the statement is executed. As with expressions, we say that a statement does something. A statement usually ends with a semicolon and represents a "step" of the program. Statements are executed in top-to-bottom order. The shortest possible statement is the null statement consisting only of the semicolon; it has no effect. In a typical program, most statements evaluate one or several expressions.

A statement is not restricted to one line of source code; on the contrary, readability often requires to break up statements into several lines of code. The compiler ignores these line breaks, as long as we do not put them at unreasonable places like in the middle of a name.

In `power8.cpp`, there are three kinds of statements.

**Expression statement.** Appending a semicolon to an expression leads to an expression statement. It evaluates the expression but does not make use of its value. This is a frequent form of statements, and in our small program, the statement

```cpp
b = b * b;
```

as well as all statements starting with `std::cin` or `std::cout` are expression statements.

**Declaration statement.** Such a statement introduces a new name into the program. This can be the name of a variable of a given type, like in the declaration statements

```cpp
int a;
```

and

```cpp
int b = a * a;
```

A declaration statement consist of a declaration and a concluding semicolon. In our program `power8.cpp`, we deal with variable declarations; they can be of the form
where $T$ is a type, $x$ is the name of the new variable, and $expr$ is an rvalue. A variable declaration is not an expression; for example, it can occur at specific places only. But when it occurs, it behaves like an expression in the sense that a declaration also has an effect and a value. Its effect is to allocate memory for the new variable at some address, and to initialize it with the value of $expr$, if present. Its value is the resulting value of the new variable. The declaration is said to define the variable.

As in the case of expression statements, a declaration statement carries out the effect of the declaration and ignores its value.

Return statement. Such a statement is of the form

\[ \text{return expression;} \]

where $expression$ is an rvalue. It only occurs within a function. The return statement evaluates $expression$, finishes the function’s computations, and puts $expression$’s value at some (temporary) address that the caller of the function can access. Abstracting from these technicalities, we simply say that the statement returns $expression$ to the caller.

We have seen only one example so far: the statement \text{return 0;} returns the value 0 (formally, the literal 0 of value 0) to the operating system which has called the main function of our program.

Figure 2 summarizes the syntactical and semantical terms that we have introduced, along with their relations. The figure emphasizes the central role that expressions play in C++.

2.1.16 The first program revisited

If you run the executable file resulting from Program 1 for a couple of input values, you will quickly notice that something is weird. For example, on the platform of the authors, the following happens:

\begin{verbatim}
Compute a^8 for a =? 15
15^8 = -1732076671.
\end{verbatim}

Obviously, the eighth power of a positive number cannot be negative, so what is going on? We will discuss this in detail in the next section, but the short explanation is that the type \texttt{int} can only deal with numbers up to a certain size. If the mathematical
result of a computation exceeds this size, the C++ result will necessarily differ from the mathematical result.

This sounds like bad news; after all, $15^8$ is not such a big number, and if we cannot even compute with numbers of this size, what can we compute at all? The good news is that the problem is easy to fix. The authors have implemented a type called ifm::integer that is capable of dealing with integers of arbitrary size (up to the memory limits, of course). Using this type is very easy, and this is one of the key strengths of C++: we simply have to replace `int` by `ifm::integer` in our program and in addition include a library that contains the definition of the new type. Here is the accordingly changed program.

1 // Program: power8_exact.cpp
CHAPTER 2. FOUNDATIONS

2 // Raise a number to the eighth power, 
3 // using integers of arbitrary size 
4
5 #include <iostream> 
6 #include <IFM/integer.h> 
7
8 int main() 
9 {
10    // input 
11    std::cout << "Compute a^8 for a =? "; 
12    ifm::integer a; 
13    std::cin >> a; 
14
15    // computation 
16    ifm::integer b = a * a; // b = a^2 
17    b = b * b; // b = a^4 
18
19    // output b * b, i.e., a^8 
20    std::cout << a << "^8 = " << b * b << ".\n"; 
21    return 0; 
22 }

Program 3: ../progs/lecture/power8_exact.cpp

Using the above program, you can compute the correct value of 15^8:
Compute a^8 for a =? 15 
15^8 = 2562890625.

But also much larger values will work (if you happen to be interested in them):
Compute a^8 for a =? 1234567 
1234567^8 = 5396563761318393964062660689603780554533710504641.

We will not discuss the type ifm::integer any further in this book, and there's no need for it, since it just works like int (except that it does not have the size limitations of int). But whenever you need (in an exercise or challenge) larger numbers, you are free to use the type ifm::integer instead of int.

2.1.17 Details

Commenting. There is a way of writing comments that are not limited to one line of code. Any text enclosed by /* (start of comment) and */ (end of comment) is ignored by the compiler. The initial comment of our program power8.cpp could also have been written as
/*
 Program: power8.cpp
*/
2.1. A FIRST C++ PROGRAM

 Raise a number to the power 8.

This mechanism may seem useful for longer comments spanning several lines of code, but the problem is that you do not immediately recognize a line in the middle of such a construction as a comment: you always have to look for the enclosing /* and */ to be sure.

Sometimes, /* and */ are used for very short comments within lines of code, like in

```cpp
c = a /* don't divide! */ b;
```

For readability reasons, we do not advocate this kind of comment, either.

Identifiers starting with an underscore. Occasionally, real-life C++ code contains “identifiers” starting with the underscore character _, although this is not allowed according to Definition [1]. The truth is that the programmer is not allowed to use such “identifiers”; they are reserved for internal use by the compiler. Compilers should issue at least a warning, when they discover such a badly formed “identifier”, but often they just let it pass.

The main function. The main function is an exceptional function in several ways. One particular specialty is that the return statement can be omitted. A main function without a return statement at the end behaves precisely as if it would end with return 0;. This definition has been made for historical reasons mostly; it is an anomaly compared to other functions (which will be discussed later). Therefore, we stick to the explicit return statement and ask you to do the same.

Using directives. It is possible to avoid all std:: prefixes through one additional line of code, a using directive. In case of power8.cpp, this would look like in Program [4].

```cpp
1 // Program: power8_using.cpp
2 // Raise a number to the eighth power.
3
4 #include <iostream>
5
6 using namespace std;
7
8 int main()
9 {
10    // input
11    cout << "Compute x^8 for x =? ";
12    int a;
13    cin >> a;
14
15    // computation
```
int b = a * a; // b = a^2
b = b * b;    // b = a^4

// output b * b, i.e., a^8
cout << a << "^8 = " << b * b << "\n";
return 0;
}

Program 4: ../progs/lecture/power8_using.cpp

The using directive is a declaration statement of the form

using namespace X;

It allows us to use all features from namespace X without qualifying them through the prefix X::. This mechanism seems quite helpful at first sight, but it has severe drawbacks that prevent us from using (let alone advocating) it in this book.

Let's start with the major drawback. Namespaces may have a large number of features (in particular, the namespace std has), with a large number of names. cin and cout are two such names from the namespace std. It is very difficult (and also not desirable) to know all these names. On the other hand, it would be good to know them in order to avoid conflicts with the names we introduce. For example, if we define a variable named cout somewhere in Program 4, we are asking for trouble: when we later use the expression cout, it is not clear whether it refers to the standard output stream, or to our variable. We can easily avoid the variable name cout, of course, but we may accidentally introduce another name that also appears in the namespace std. The unfortunate consequence is that in some expression of our program, this name might not refer to the feature we introduced, but to a feature of the same name from the standard library. We may end up silently using a feature from the standard library that we don't even know and that we never intended to use. The resulting strange behavior of our program can be very difficult to track down.

In the original program power8.cpp, introducing a name cout (or any other name also appearing in namespace std) does not cause any harm: without the std:: qualification, it can never “accidentally” refer to something from the standard library.

Here is the second drawback of using directives. A large program contains many names, and in order to keep track of, it is desirable that the name “tells” us where it comes from: is it a name we have introduced, or does it come from a library? If so, from which one? With using directives, we lose that information, meaning that the program becomes less readable and more difficult to maintain.

2.1.18 Goals

Dispositional. At this point, you should ...
1) understand the basic syntactical and semantical terms of C++, in particular expression, operator, statement, lvalue, rvalue, literal, variable, and constant;
2) understand syntax and semantics of the program power8.cpp.

Operational. In particular, you should be able to ...

(G1) describe the basic syntactical and semantical terms of C++ (as listed above) in your own words and give examples;
(G2) tell whether a given character sequence is an identifier;
(G3) tell whether a given character sequence is a simple expression, as defined below;
(G4) find out whether a given simple expression is an lvalue or an rvalue;
(G5) evaluate a given simple expression;
(G6) check whether a program of complexity similar to power8.cpp correctly uses the const keyword, and whether it follows the Const Guideline (meaning that it is const-correct).
(G7) read and write programs with functionality similar to power8.cpp.

A simple expression is an expression which only involves int-literals, identifiers, the binary multiplication operator *, the assignment operator, and parentheses.

2.1.19 Exercises

Exercise 1 Which of the following character sequences are not C++ identifiers, and why not? (G2)
(a) identifier (b) int (c) x_i (d) 4x__
(e) A99_ (f) _tmp (g) T# (h) x12b

Exercise 2 Which of the following character sequences are not C++ expressions, and why not? Here, a and b are variables of type int. (G3)
(a) 1*(2*3) (b) a=(b=5) (c) 1=a (d) (a=1)
(e) (a=5)*(b=7) (f) (1 (g) (a=b)*(b=5) (h) (a+3)=(b*5)

Exercise 3 For all of the expressions that you have identified in Exercise 2 decide whether these are lvalues or rvalues, and explain your decisions. (G4)

Exercise 4 Determine the values of the expressions that you have identified in Exercise 2 and explain how these values are obtained. Which of these values are unspecified and can therefore not be determined uniquely? (G5)

Exercise 5 Which of the following (rather stupid) programs are syntactically incorrect (w.r.t. the usage of const), and why? Among the correct ones, which programs do not adhere to the Const Guideline of Section 2.1.9 and why? (G6).
a) int main ()
{
    const int a = 5;
    int b = a;
    b = b*2;
    return 0;
}

b) #include <iostream>
int main ()
{
    const int a = 5;
    std::cin >> a;
    std::cout << a + 5;
    return 0;
}

c) #include <iostream>
int main ()
{
    const int a;
    int b;
    std::cin >> b;
    std::cout << a;
    return 0;
}

d) int main ()
{
    const int a = 5;
    int b = 2*a;
    int c = 2*b;
    b = b*b;
    return 0;
}

e) int main ()
{
    const int a = 5;
    const int b = (a = 6);
    return 0;
}
2.1. A FIRST C++ PROGRAM

f) int main ()
    {
        const int a = 5;
        a = 5;
        return 0;
    }

g) #include <iostream>
    int main ()
    {
        int a = 5;
        a = a*a;
        int b = a;
        b = b*b;
        const int c = b;
        std::cout << c*c;
        return 0;
    }

Exercise 6 What is the smallest natural number that is divisible by all numbers between 2 and

a) 10 ?

b) 20 ?

c) 30 ?

The result is also known as the least common multiple of the respective numbers.

Note: This exercise does not require you to write a program, but you may use a program to help you in the computations.

Exercise 7 Write a program multhree.cpp that reads three integers a, b, c from standard input and outputs their product abc.

Exercise 8 Write a program power20.cpp that reads an integer a from standard input and outputs \( a^{20} \) using at most five multiplications.

Exercise 9 During an electronic transmission, the following C++ program got garbled. As you can see, the layout got messed up, but at the same time, some errors got introduced as well.

#include <iostream>
int main[] {int a;int b;int c;std::cin >> a;
cin >> b;c = a * b;std::cout << c*c;return 0;}
a) Write the program down in a well-formatted way.

b) The program contains two syntax errors. Fix them! What does the fixed program do?

c) The (fixed) program contains a number of composite expressions. List them all, and decide for each composite expression whether it is an rvalue or an lvalue. Recall that a composite expression may consist of primary expressions, but also of other composite expressions.

d) Add sensible comments to the program; most notably, there should be a comment in the beginning that says what the program does.

e) Move the variable declarations to their logical places (immediately before first use), and make the program const-correct.

Exercise 10 Write a program age_verification.cpp that defines the current year (e.g. 2009) as a constant, and then outputs the age groups that are not allowed to buy alcohol. (If you are a manager at Coop, you can now post this at every checkout). For the year 2009, the output should be

No alcohol to people born in the years 1992 - 2009!
For people born in 1991, check the id!

2.1.20 Challenges

Exercise 11 The obvious “slow” method for computing the eighth power of an integer a needs seven multiplications. Program [4] requires only three, and we believe that this should be faster. The goal of this challenge is to find out: how much faster? For example, if we compute the eighth power of a 10,000-digit number using both methods, what will be the difference in runtime? Using the type int, though, we cannot correctly compute with 10,000-digit numbers (as you will easily notice when you start power8.cpp with somewhat larger inputs, see Section [2.1.16]). For this reason, you should use the type ifm::integer for your computations.

Write two programs, power8_slow.cpp and power8_fast.cpp that compute the eighth power of an integer with 7 and 3 multiplications, respectively (over the exact type ifm::integer). Since we want to measure runtimes, there should be no output (you don’t want to read it, anyway). In order to be able to use the same large inputs for both programs, it is beneficial to have the programs read the input from a file. For example, if you have a file power8.dat with contents 1234567, you can tell the program power8_exact.cpp to read its input from this file by starting it with the command

./power8_exact < power8.dat
2.1. A FIRST C++ PROGRAM

Now that you have both programs, create an input file power8.dat, and fill it with larger and larger numbers (each time doubling the number of digits, for example). Then measure the times taken by each of the programs power8_slow.cpp and power8_fast.cpp on these inputs. You can simply do this using your watch (for sufficiently many digits both programs will be slow enough), or you can start the programs like this:

```
time ./power8_fast < power8.dat
```

This command will run the program and afterwards output the number of seconds that it took (the first number, the one ending in u, is the relevant one).

What do you observe? Is power8_fast.cpp more than twice as fast as power8_slow.cpp (this is what you might expect from the number of multiplications)? Or do you observe a speedup factor quite different from 2? And is this factor stable as the input numbers get larger?

Whatever you observe, try to explain your observations!

Exercise 12 Let \( L(n) \) be the smallest number of multiplications that are needed in order to compute the \( n \)-th power \( a^n \) of an integer \( a \). Since \( L(n) \) may depend on what we consider as a “computation”, we make \( L(n) \) well-defined by restricting to the following kind of computations. Let \( a_0 \) denote the input number. A computation consists of \( t \) steps, where \( t \) is some natural number, and step \( i, 1 \leq i \leq t \) has the form

\[
a_i = a_j \cdot a_k
\]

with \( j, k < i \). The computation is correct if \( a_t = a^n \). For example, to compute \( a^8 \) in three steps (three multiplications) as in power8.cpp, we can use the computation

\[
a_1 = a_0 \cdot a_0 \\
a_2 = a_1 \cdot a_1 \\
a_3 = a_2 \cdot a_2
\]

Now, \( L(n) \) is defined as the smallest value of \( t \) such that there exists a correct \( t \)-step computation for \( a^n \).

a) In the above model of computation, prove that for all \( n \geq 1 \),

\[
\lambda(n) \leq L(n) \leq \lambda(n) + \nu(n) - 1,
\]

where \( \lambda(n) \) is one less than the number of significant bits of \( n \) in the binary representation of \( n \) (see Section 2.2.5), and \( \nu(n) \) is the number of 1’s in the binary representation of \( n \). For example, the binary representation of 20 is 10100, and hence \( \lambda(20) = 4 \) and \( \nu(20) = 2 \), resulting in \( L(20) \leq 5 \).

b) Either prove that the upper bound in a) is always best possible, or find a value \( n^* \) such that \( L(n^*) < \lambda(n^*) + \nu(n^*) - 1 \).
2.2 Integers

Die ganzen Zahlen hat der liebe Gott gemacht, alles andere ist Menschenwerk.

Leopold Kronecker, in a lecture to the Berliner Naturforscher-Versammlung (1886)

This section discusses the types int and unsigned int for representing integers and natural numbers, respectively. You will learn how to evaluate arithmetic expressions over both types. You will also understand the limitations of these types, and—related to this—how their values can be represented in the computer’s memory.

Here is our next C++ program. It asks the user to input a temperature in degrees Celsius, and outputs it in degrees Fahrenheit. The conversion is defined by the following formula.

\[
\text{Degrees Fahrenheit} = \frac{9 \cdot \text{Degrees Celsius}}{5} + 32.
\]

Program 5: ../progs/lecture/fahrenheit.cpp

If you try out the program on the input of 15 degrees Celsius, you will get the following output.

15 degrees Celsius are 59 degrees Fahrenheit.
This output is produced when the expression statement in lines 14–15 of the program is executed. Here we focus on the evaluation of the arithmetic expression

\[ 9 \times \text{celsius} / 5 + 32 \]

in line 15. This expression contains the primary expressions 9, 5, 32, and celsius, where celsius is a variable of type int. This fundamental type is one of the arithmetic types in C++.

**Literals of type int.** 9, 5 and 32 are decimal literals of type int, with their values immediately apparent. Decimal literals of type int consist of a sequence of digits from 0 to 9, where the first digit must not be 0. The value of a decimal literal is the decimal number represented by the sequence of digits. There are no literals for negative integers. You can get value $-9$ by writing $-9$, but this is a composite expression built from the unary subtraction operator (Section 2.2.4) and the literal 9.

### 2.2.1 Associativity and precedence of operators

The evaluation of an expression is to a large extent governed by the associativities and precedences of the involved operators. In short, associativities and precedences determine the logical parentheses in an expression that is not, or only incompletely, parenthesized. We have already touched associativity in connection with the output operator in Section 2.1.14.

C++ allows incompletely parenthesized expressions in order to save parentheses at obvious places. This is like in mathematics, where we write $3 + 4 \cdot 5$ when we mean $3 + (4 \cdot 5)$. We also write $3 + 4 + 5$, even though it is not a priori clear whether this means $(3 + 4) + 5$ or $3 + (4 + 5)$. Here, the justification is that addition is associative, so it does not matter which variant we mean.

The price to pay for less parentheses is that we have to know the logical parentheses. But this is a moderate price, since the two rules that are used most frequently are quite intuitive and easy to remember. Also, there is always the option of explicitly adding parentheses in case you are not sure where C++ would put them. Let us start with the two essential rules for arithmetic expressions.

| Arithmetic Evaluation Rule 1: Multiplicative operators have higher precedence than additive operators. |

The expression $9 \times \text{celsius} / 5 + 32$ involves the multiplication operator $\times$, the division operator $/$, and the addition operator $+$. All three are binary operators. In C++ as in mathematics, the multiplicative operators $\times$ and $/$ have higher precedence than the additive operators $+$ and $-$. We also say that multiplicative operators bind more strongly than additive ones. This means, our expression contains the logical parentheses $(9 \times \text{celsius} / 5) + 32$: it is a composite expression built from the addition operator and its operands $9 \times \text{celsius} / 5$ and 32.
In mathematics, it does not matter how the sub-expression \(9 \times \text{celsius} / 5\) is parenthesized. But in C++, it is done from \textit{left to right}, that is, the two leftmost sub-expressions are grouped together. This is a consequence of the fact that the binary arithmetic operators are defined to be \textit{left} associative. The expression \(9 \times \text{celsius} / 5\) is therefore logically parenthesized as \((9 \times \text{celsius}) / 5\), and our original expression has to be read as

\[
((9 \times \text{celsius}) / 5) + 32
\]

\textbf{Arithmetic Evaluation Rule 2: Binary arithmetic operators are left associative.}

**Identifying the operators in an expression.** There is one issue we haven’t discussed yet, namely that different C++ operators may have the same token. For example, \(-\) can be a binary operator as in \(3 - 4\), but it can also be a unary operator as in \(-5\). Which one is meant must be inferred from the context. Usually, this is clear, and in cases where it is not (but also in other cases), it is probably a good idea to add some extra parentheses to make the expression more readable (see also the Details section below).

Let us consider another concrete example, the expression \(-3 - 4\). It is clear that the first \(-\) must be unary (there is no left hand side operand), while the second one is binary (there are operands on both sides). But is this expression logically parenthesized as \((-3 - 4)\), or as \((-3) - 4\)? Since we get different values in both cases, we better make sure that we know the answer.

The correct logical parentheses are

\[
((-3) - 4),
\]

so the value of the expression \(-3 - 4\) is \(-7\). This follows from the third most important rule for arithmetic expressions.

**Arithmetic Evaluation Rule 3: Unary operators + and - have higher precedence than their binary counterparts.**

By using (explicit) parentheses as in \(9 \times (\text{celsius} + 5) \times 32\), precedences can be overruled. To get the logical parentheses for such a partially parenthesized expression, we apply the rules from above, considering the already parenthesized parts as operands. In the example, this leads to the logical parentheses \((9 \times (\text{celsius} + 5)) \times 32\).

The Details section discusses how to parenthesize a general expression involving arbitrary C++ operators, using their arities, precedences and associativities.

### 2.2.2 Expression trees

In every composite expression, the logical parentheses determine a unique “top-level” operator, namely the one that appears within the smallest number of parentheses. The
expression is then a composite expression, built from the top-level operator and its operands that are again expressions.

The structure of an expression can nicely be visualized in the form of an expression tree. In Figure 3, the expression tree for the expression \( 9 \times \text{celsius} / 5 + 32 \) is shown.

![Figure 3: An expression tree for \( 9 \times \text{celsius} / 5 + 32 \) and its logical parentheses \((9 \times \text{celsius}) / 5) + 32\). Nodes are labeled from one to seven.](image)

How do we get this tree? The expression itself defines the root of the tree, and the operands of the top-level operator become the root’s children in the tree. Each operand then serves as the root of another subtree. When we reach a primary expression, it defines a leaf in the tree, with no further children.

### 2.2.3 Evaluating expressions

From an expression tree we can easily read off the possible evaluation sequences for the arithmetic expression. Such a sequence contains all sub-expressions occurring in the tree, ordered by their time of evaluation. For this sequence to be valid, we have to make sure that we evaluate an expression only after the expressions corresponding to all its children have been evaluated. By looking at Figure 3 this becomes clear: before evaluating \( 9 \times \text{celsius} \), we have to evaluate 9 and \( \text{celsius} \), otherwise, we don’t have enough information to perform the evaluation.

When we associate the evaluation sequence with the corresponding sequence of nodes in the tree, a valid node sequence topologically sorts the tree. This means that every node in the sequence occurs only after all its children have occurred. In Figure 3, for example, the node sequence \( 1, 2, 5, 3, 6, 4, 7 \) induces a valid evaluation sequence. Assuming that the variable \( \text{celsius} \) has value 15, we obtain the following evaluation sequence. (In each step, the sub-expression to be evaluated next is marked by a surrounding box).
The sequence \((1, 2, 3, 4, 5, 6, 7)\) is another valid node sequence, inducing a different evaluation sequence; the resulting value of 59 is the same. There are much more evaluation sequences, of course, and it is unspecified by the C++ standard which one is to be used.

In our small example, all possible evaluation sequences will result in value 59, but it is also not hard to write down expressions whose values and effects depend on the evaluation sequence being chosen (see Exercise 2(g), Exercise 15(h), and the Details section below). A program that contains such an expression might exhibit unspecified behavior. But through good programming style, this issue is easy to avoid, since it typically only occurs when one tries to squeeze too much functionality into a single line of code.

### 2.2.4 Arithmetic operators on the type int

In the program fahrenheit.cpp, we have already encountered the multiplicative operators \(*\) and \(/\), as well as the binary addition operator \(+\). Its obvious counterpart is the binary subtraction operator \(-\).

Table 1 lists arithmetic operators (and the derived assignment operators) that are available for the type \(\text{int}\), with their arities, precedences and associativities. The actual numbers that appear in the precedence column are not relevant: it is the order among precedences that matters.

Let us discuss the functionalities of these operators in turn, where \(*\), \(+\) and \(-\) are self-explanatory. But already the division operator requires a discussion.

**The division operator.** According to the rules of mathematics, we could replace the expression

\[
9 \ast \text{celsius} \div 5 + 32
\]

by the expression

\[
9 \div 5 \ast \text{celsius} + 32
\]

without affecting its value and the functionality of the program fahrenheit.cpp. But if we run the program with the latter version of the expression on the input of 15 degrees Celsius, we get the following output:

\[
\begin{align*}
9 \ast \text{celsius} \div 5 + 32 & \rightarrow^1 9 \ast \boxed{\text{celsius}} \div 5 + 32 \\
& \rightarrow^2 9 \ast 15 \div 5 + 32 \\
& \rightarrow^5 135 \div 5 + 32 \\
& \rightarrow^3 135 \div 5 + 32 \\
& \rightarrow^6 27 + 32 \\
& \rightarrow^4 27 + 32 \\
\rightarrow^7 59
\end{align*}
\]
Table 1: Arithmetic and assignment operators for the type `int`. Each increment or decrement operator expects an lvalue. The composite expression is an lvalue (pre-increment and pre-decrement), or an rvalue (post-increment and post-decrement). Each assignment operator expects an lvalue as left operand and an rvalue as right operand; the composite expression is an lvalue. All other operators involve rvalues only and have no effects.
15 degrees Celsius are 47 degrees Fahrenheit.

This result is fairly different from our previous (and correct) result of 59 degrees Fahrenheit, so what is going on here? The answer is that the binary division operator / on the type int implements the integer division, in mathematics denoted by div. This does not correspond to the regular division where the quotient of two integers is in general a non-integral rational number.

The modulus operator. The remainder of the integer division can be obtained with the binary modulus operator %, in mathematics denoted by mod. The mathematical rule

\[ a = (a \div b)b + a \mod b \]

also holds in C++: for example, if \( a \) and \( b \) are variables of type int, the value of \( b \) being non-zero, the expression

\[(a / b) * b + a \mod b\]

has the same value as \( a \). The modulus operator is considered as a multiplicative operator and has the same precedence (14) and associativity (left) as the other two multiplicative operators \(*\) and \(/\).

If both \( a \) and \( b \) have non-negative values, then \( a \mod b \) has a non-negative value as well. This implies that the integer division rounds down in this case. If (at least) one of \( a \) or \( b \) has a negative value, it is implementation defined whether division rounds up or down. Note that by the identity \( a = (a / b) * b + a \mod b \), the rounding mode for division also determines the functionality of the modulus operator. If \( b \) has value 0, the values of \( a / b \) and \( a \mod b \) are undefined.

Coming back to our example (and taking precedences and associativities into account), we get the following valid evaluation sequence for our alternative Celsius-to-Fahrenheit conversion.

\[ 9 / 5 \times celsius + 32 \rightarrow 1 \times celsius + 32 \]
\[ \rightarrow 15 + 32 \]
\[ \rightarrow 47 \]

Here we see the “error” made by the integer division: \( 9 / 5 \) has value 1.

Unary additive operators. We have already touched the unary - operator, and this operator does what one expects: the value of the composite expression -expr is the negative of the value of expr. There is a unary + operator, for completeness, although its “functionality” is non-existing: the value of the composite expression +expr is the same as the value of expr.
Increment and decrement operators. Each of the tokens ++ and -- is associated with two distinct unary operators that differ in precedence and associativity.

The pre-increment ++ and the pre-decrement -- are right associative. For a unary operator, this means that the argument appears to the right of the operator token. The effect of the composite expressions ++\(expr\) and --\(expr\) is to increase (decrease, respectively) the value of \(expr\) by 1. Then, the object referred to by \(expr\) is returned. For this to make sense, \(expr\) has to be an lvalue. We also say that pre-increment is ++ in prefix notation, and similarly for --.

The post-increment ++ and the post-decrement -- are left associative. As before, the effect of the composite expressions \(expr++\) and \(expr--\) is to increase (respectively decrease) the value of \(expr\) by 1, and \(expr\) has to be an lvalue for this to work. The return value, though, is an rvalue corresponding to the old value of \(expr\) before the increment or decrement took place. We also say that post-increment is ++ in postfix notation, and similarly for --.

The difference between the increment operators in pre- and postfix notation is illustrated in the following example program.

```cpp
#include <iostream>
int main() {
    int a = 7;
    std::cout << ++a << "\n"; // outputs 8
    std::cout << a++ << "\n"; // outputs 8
    std::cout << a << "\n"; // outputs 9
    return 0;
}
```

You may argue that the increment and decrement operators are superfluous, since their functionality can be realized by combining the assignment operator (Section 2.1.14) with an additive operator. Indeed, if \(a\) is a variable, the expression \(++a\) is equivalent in value and effect to the expression \(a = a + 1\). There is one subtlety, though: if \(expr\) is a general lvalue, \(++expr\) is not necessarily equivalent to \(expr = expr + 1\). The reason is that in the former expression, \(expr\) is evaluated once only, while in the latter, it is evaluated twice. If \(expr\) has an effect, this can make a difference.

On the other hand, this subtlety is not the reason why increment and decrement operators are so popular and widely used in C++. The truth is that incrementing or decrementing values by 1 are such frequent operations in typical C++ code that it pays off to have shortcuts for them.

Prefer pre-increment over post-increment. The statements ++\(i\); and \(i++\); are obviously equivalent, as their effect is the same and the value of the expression is not used. You can exchange them with each other arbitrarily without affecting the behavior of the surrounding program. Whenever you have this choice, you should opt for the pre-increment operator. Pre-increment is the simpler operation because the value of ++\(i\) can simply be read off the variable \(i\). In contrast, the post-increment has to 'remember' the original
value of $i$. As pre-increment is simpler, it also tends to be more efficient.

**Remark:** We write "pre-increment tends to be more efficient" because in many cases the compiler realizes when the value of an expression is not used. In such a case, the compiler may choose on its own to replace the post-increment in the source code by a "pre-increment" in machine language as an optimization. However, there is absolutely no benefit in choosing a post-increment where a pre-increment would do as well. In this case, you should take the burden from the compiler and optimize by yourself.

Also, post-increment and post-decrement are the only unary C++ operators that are left associative. This makes their usage appear somewhat counterintuitive.

**Why C++ should rather be called ++C.** The language C++ is a further development of the language C. And indeed, you can read the effect of the "expression" C++ as "take C one step further". But the name C++ is still a misnomer, because the value of the "expression" C++ is the plain old C, after all. A better name would be ++C: the "value" of this is really the new language created from taking C one step further. Bjarne Stroustrup, the designer of C++, writes that "Connoisseurs of C semantics find C++ inferior to ++C".

**Assignment operators.** The assignment operator $=$ is available for all types, see Section 2.1.14. But there are specific operators that combine the arithmetic operators with an assignment. These are the binary operators $+=,-=,*=,/=$ and $%=$. The expression $expr1 += expr2$ has the effect of adding the value of $expr2$ (an rvalue) to the value of $expr1$ (an lvalue). The object referred to by $expr1$ is returned. This is a generalization of the pre-increment: the expression $++expr$ is equivalent to $expr += 1$. As before, $expr1 += expr2$ is not equivalent to $expr1 = expr1 + expr2$ in general, since the latter expression evaluates $expr1$ twice.

The operators $-=$, $*=$, $/=$ and $%=$ work in the same fashion, based on the subtraction, multiplication, division, and modulus operator, respectively.

All the assignment operators have precedence 4, i.e. they bind more weakly than the other arithmetic operators. This is quite intuitive: $a=b*c-d$, say, means $a=(b*c-d)$.

**2.2.5 Value range**

A variable of type `int` is associated with a fixed number of memory cells, and therefore also with a fixed number of bits, say $b$. We call this a $b$-bit representation.

Such a representation implies that an object of type `int` can assume only finitely many different values. Since every bit can independently have two states, the maximum number of representable values is $2^b$, and the actual value range is defined as the set

$$\{-2^{b-1}, -2^{b-1} + 1, \ldots, -1, 0, 1, \ldots, 2^{b-1} - 1\} \subset \mathbb{Z}$$

of $2^b$ numbers. The C++ standard does not prescribe this, but any different choice of value range would be somewhat unreasonable, given other requirements imposed by the
standard. You can find out the smallest and largest int values on your platform, using the library limits. The corresponding code is given in Program 6.

```cpp
1 // Program: limits.cpp
2 // Output the smallest and the largest value of type int.
3
4 #include <iostream>
5 #include <limits>
6
7 int main()
8 {
9     std::cout << "Minimum int value is "
10         << std::numeric_limits<int>::min() << ".\n"
11     << "Maximum int value is "
12         << std::numeric_limits<int>::max() << ".\n";
13     return 0;
14 }
```

Program 6: ../progs/lecture/limits.cpp

When you run the program limits.cpp on a 32-bit system, you may get the following output.

Minimum int value is -2147483648.
Maximum int value is 2147483647.

Indeed, as $2^{31} - 1$, you can deduce that the number of bits used to represent an int value on your platform is 32. At this point, you are not supposed to understand expressions like std::numeric_limits<int>::min() in detail, but we believe that you get their idea. We cannot resist to note in passing that $2^{31} - 1$ is the number from Mersenne's conjecture that Euler has proved to be prime in 1772, see Section 1.1.

It is clear that the arithmetic operators (except the unary + and the binary / and \%) cannot work exactly like their mathematical counterparts, even when their arguments are restricted to representable int values. The reason is that the values of composite expressions constructed from these operators can under- or overflow the value range of the type int. The most obvious such example is the expression $2^{31}483647+1$. As we have just seen, its mathematically correct value of 2147483648 may not be representable over the type int on your platform, in which case you will inevitably get some other value.

Such under- and overflows are a severe problem in many practical applications, but it would be an even more severe problem not to know that they can occur.
2.2.6 The type unsigned int

An object of type int can have negative values, but often we only work with natural numbers. (For us, the set \( \mathbb{N} \) of natural numbers starts with 0, \( \mathbb{N} = \{0, 1, 2, \ldots \} \).) Using a type that represents only non-negative values allows to extend the range of positive values without using more bits. C++ provides such a type, it is called unsigned int. On this type, we have all the arithmetic operators we also have for int, with the same arities, precedences and associativities. Given a \( b \)-bit representation, the value range of unsigned int is the set

\[
\{0, 1, \ldots, 2^b - 1\} \subseteq \mathbb{N}
\]

of \( 2^b \) natural numbers. Indeed, when you replace all occurrences of int by unsigned int in the program limits.cpp, it may produce the following output.

Minimum value of an unsigned int object is 0.

Maximum value of an unsigned int object is 4294967295.

Literals of type unsigned int look like literals of type int, followed by either the letter u or U. For example, 127u and 0U are literals of type unsigned int, with their values immediately apparent.

2.2.7 Mixed expressions and conversions

Expressions may involve sub-expressions of type int and of type unsigned int. For example 17+17u is a legal arithmetic expression, but what are its type and value? In such mixed expressions, the operands are implicitly converted to the more general type. By the C++ standard, the more general type is unsigned int. Therefore, the expression 17+17u is of type unsigned int and gets evaluated step by step as

\[
17+17u \rightarrow 17u+17u \rightarrow 34u
\]

This might be somewhat confusing, since in mathematics, it is just the other way around: \( \mathbb{Z} \) (the set of integers) is more general than \( \mathbb{N} \) (the set of natural numbers). We are not aware of any deeper justification for the way it is done in C++, but at least the conversion is well-defined:

Non-negative int values are “converted” to the same value of type unsigned int; negative int values are converted to the unsigned int value that results from (mathematically) adding \( 2^b \). This rule establishes a bijection between the value ranges of int and unsigned int.

Implicit conversions in the other direction may also occur but are not always well-defined. Consider for example the declarations

```cpp
int a = 3u;
int b = 4294967295u;
```

The value of a is 3, since this value is in the range of the type int. But if we assume the 32-bit system from above, the value of b is implementation defined according to the C++ standard, since the literal 4294967295 is outside the range of int.
2.2.8 Binary representation

Assuming \( b \)-bit representation, we already know that the type int covers the values

\[-2^{b-1}, \ldots, 2^{b-1} - 1,
\]

while unsigned int covers

\[0, \ldots, 2^b - 1.
\]

In this subsection, we want to take a closer look at how these values are represented in memory, using the \( b \) available bits. This will also shed more light on some of the material in the previous subsection.

The binary expansion of a natural number \( n \in \mathbb{N} \) is the sum

\[n = \sum_{i=0}^{\infty} b_i 2^i,
\]

where the \( b_i \) are uniquely determined coefficients from \( \{0, 1\} \), with only finitely many of them being nonzero. For example,

\[13 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3.
\]

The sequence of the \( b_i \) in reverse order is called the binary representation of \( n \). The binary representation of 13 is 1101, for example.

Conversion decimal \( \rightarrow \) binary. The identity

\[n = \sum_{i=0}^{\infty} b_i 2^i = b_0 + \sum_{i=1}^{\infty} b_i 2^i = b_0 + \sum_{i=0}^{\infty} b_{i+1} 2^{i+1} = b_0 + 2 \sum_{i=0}^{\infty} b_{i+1} 2^i
\]

provides a simple algorithm to compute the binary representation of a given decimal number \( n \in \mathbb{N} \). The least significant coefficient \( b_0 \) of the binary expansion of \( n \) is \( n \bmod 2 \). The other coefficients \( b_i, i \geq 1 \), can subsequently be extracted by applying the same technique to \( n' = (n - b_0)/2 \).

For example, for \( n = 14 \) we get \( b_0 = 14 \bmod 2 = 0 \) and \( n' = (14 - 0)/2 = 7 \). We continue with \( n = 7 \) and get \( b_1 = 7 \bmod 2 = 1 \) and \( n' = (7 - 1)/2 = 3 \). For \( n = 3 \) we get \( b_2 = 3 \bmod 2 = 1 \) and \( n' = (3 - 1)/2 = 1 \) which leaves us with \( n = b_3 = 1 \). In summary, the binary representation of 14 is \( b_3 b_2 b_1 b_0 = 1110 \).

Conversion binary \( \rightarrow \) decimal. To convert a given binary number \( b_k \ldots b_0 \) into decimal representation, we can once again use the identity from above.

\[
\sum_{i=0}^{k} b_i 2^i = b_0 + 2 \sum_{i=0}^{k-1} b_{i+1} 2^i = \ldots = b_0 + 2(b_1 + 2(b_2 + 2(\cdots + 2b_k)\ldots))
\]
For example, to convert the binary number \(b_4b_3b_2b_1b_0 = 10100\) into decimal representation, we compute
\[
(((b_4 \cdot 2 + b_3) \cdot 2 + b_2) \cdot 2 + b_1) \cdot 2 + b_0 = (((1 \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 0) \cdot 2 + 0 = 20.
\]

**Representing unsigned int values.** Since every unsigned int value \(n \in \{0, \ldots, 2^b - 1\}\) has a binary representation of length exactly \(b\) (filling up with leading zeros), this binary representation is a canonical format for storing \(n\) using the \(b\) available bits. Like the value range itself, this storage format is not explicitly prescribed by the C++ standard, but hardly anything else makes sense in practice. As there are \(2^b\) unsigned int values, and the same number of \(b\)-bit patterns, each pattern encodes one value. For \(b = 3\), this looks as follows.

<table>
<thead>
<tr>
<th>(n)</th>
<th>representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td></td>
</tr>
<tr>
<td>1 001</td>
<td></td>
</tr>
<tr>
<td>2 010</td>
<td></td>
</tr>
<tr>
<td>3 011</td>
<td></td>
</tr>
<tr>
<td>4 100</td>
<td></td>
</tr>
<tr>
<td>5 101</td>
<td></td>
</tr>
<tr>
<td>6 110</td>
<td></td>
</tr>
<tr>
<td>7 111</td>
<td></td>
</tr>
</tbody>
</table>

**Representing int values.** A common way of representing int values using the same \(b\) bits goes as follows. If the value \(n\) is non-negative, we store the binary representation of \(n\) itself—a number from
\[
\{0, \ldots, 2^{b-1} - 1\}.
\]
That way we use all the \(b\)-bit patterns that start with 0.

If the value \(n\) is negative, we store the binary representation of \(n + 2^b\), a number from
\[
\{2^{b-1}, \ldots, 2^b - 1\}.
\]
This yields the missing \(b\)-bit patterns, the ones that start with 1. For \(b = 3\), the resulting representations are
This is called the two's complement representation. In this representation, adding two int values \( n \) and \( n' \) is very easy: simply add the representations according to the usual rules of binary number addition, and ignore the overflow bit (if any). For example, to add \(-2\) and \(-1\) in case of \( b = 3 \), we compute

\[
\begin{array}{c}
110 \\
+ \quad 111 \\
\hline
1101
\end{array}
\]

Ignoring the leftmost overflow bit, this gives \( 101 \), the representation of the result \(-3\) in two's complement. This works since the binary number behind the encoding of \( n \) is either \( n \) or \( n + 2^b \). Thus, when we add the binary numbers for \( n \) and \( n' \), the result is congruent to \( n + n' \) modulo \( 2^b \) and therefore agrees with \( n + n' \) in the \( b \) rightmost bits.

Using the two's complement representation we can now better understand what happens when a negative int value \( n \) gets converted to type unsigned int. The standard specifies that for this, \( n \) has to be incremented by \( 2^b \). But under the two's complement, the negative int value \( n \) and the resulting positive unsigned int value \( n + 2^b \) have the same representation! This means that the conversion is purely conceptual, and no actual computation takes place.

The C++ standard does not prescribe the use of the two's complement, but the rule for conversion from int to unsigned int is clearly motivated by it.

### 2.2.9 Integral types

There is a number of other fundamental types to represent signed and unsigned integers, see the Details section. These types may differ from int and unsigned int with respect to their value range. All these types are called integral types, and for each of them, all the operators in Table 1 (Page 51) are available, with the same arities, precedences, associativities and functionalities (up to the obvious limits dictated by the respective value ranges).
2.2.10 Details

Literals. There are also non-decimal literals of type int. An octal literal starts with the digit 0, followed by a sequence of digits from 0 to 7. The value is the octal number represented by the sequence of digits following the leading 0. For example, the literal 011 has value $9 = 1 \cdot 8^1 + 1 \cdot 8^0$.

Hexadecimal literals start with 0x, followed by a sequence of digits from 0 to 9 and letters from A to F (representing the hexadecimal digits of values 10, \ldots, 15). The value is the hexadecimal number represented by the sequence of digits and letters following the leading 0x. For example, the literal 0x1F has value $31 = 1 \cdot 16^1 + 15 \cdot 16^0$.

Logically parenthesizing a general expression. Given an expression that consists of a sequence of operators and operands, we want to deduce the logical parentheses. For each operator in the sequence, we know its arity, its precedence (a number between 1 and 18, see Table 1 on Page 51 for the arithmetic operators), and its associativity (left or right). In case of a unary operator, the associativity specifies on which side of the operator its operand is to be found.

Let us consider the following abstract example to emphasize that what we do here is completely general and not restricted to arithmetic expressions.

Here is how the parentheses are obtained: for each operator, we identify its leading operand, defined as the left hand side operand for left associative operators, and as the right hand side operand otherwise. The leading operand for op$_1$ includes everything to the relevant side between op$_1$ and the next operator of lower precedence than op$_1$. In other words, everything in between these two operators is “grabbed” by the “stronger” operator.

In our example, the leading operand of op$_3$ is the subsequence $x_2 \ op_2 \ x_3$ to the left of op$_3$, since the next operator of lower precedence to the left of op$_3$ is op$_1$.

In the case of binary operators, we also find the secondary operand, the one to the other side of the leading operand. The secondary operand for op$_1$ includes everything to the relevant side between op$_1$ and the next operator of the same or lower precedence than op$_1$. The only difference to the leading operand rule is that the secondary operand already ends when an operator of the same precedence appears.

According to this definition, the secondary operand of op$_3$ is op$_4 \ x_4$ in our example.

Finally, we put a pair of parentheses around the subsequence corresponding to the leading operand, the operator itself, and the secondary operand (if any).

Here is the table for our example again, enhanced with the subsequences of all four operators that are put in parentheses according to the rules just described.
expression | $x_1$ | $\mathbf{op}_1$ | $x_2$ | $\mathbf{op}_2$ | $x_3$ | $\mathbf{op}_3$ | $x_4$ | $\mathbf{op}_4$ | $x_4$
-----|-----|-----|-----|-----|-----|-----|-----|-----|-----
arity | 2 | 2 | 2 | 1
precedence | 4 | 13 | 13 | 16
associativity | r | l | l | r

$\mathbf{op}_1$ | ( | $x_1$ | $\mathbf{op}_1$ | $x_2$ | $\mathbf{op}_2$ | $x_3$ | $\mathbf{op}_3$ | $x_4$ | $\mathbf{op}_4$ | $x_4$ )
$\mathbf{op}_2$ | ( | $x_2$ | $\mathbf{op}_2$ | $x_3$ )
$\mathbf{op}_3$ | ( | $x_2$ | $\mathbf{op}_2$ | $x_3$ | $\mathbf{op}_3$ | $\mathbf{op}_4$ | $x_4$ )
$\mathbf{op}_4$ | ( | $\mathbf{op}_4$ | $x_4$ )

Now we simply put together all parentheses that we have obtained, taking their multiplicities into account. In our example we get the expression

$$( x_1 \mathbf{op}_1 ( ( x_2 \mathbf{op}_2 x_3 ) \mathbf{op}_3 ( \mathbf{op}_4 x_4 ))) .$$

By some magic, this worked out, and we have a fully parenthesized expression (the outer pair of parentheses can be dropped again, of course). But note that we cannot expect such nice behavior in general. Consider the following example.

expression | $x_1$ | $\mathbf{op}_1$ | $x_2$ | $\mathbf{op}_2$ | $x_3$
-----|-----|-----|-----|-----|-----
arity | 2 | 2
precedence | 13 | 13
associativity | r | l

$\mathbf{op}_1$ | ( | $x_1$ | $\mathbf{op}_1$ | $x_2$ | $\mathbf{op}_2$ | $x_3$ )
$\mathbf{op}_2$ | ( | $x_1$ | $\mathbf{op}_1$ | $x_2$ | $\mathbf{op}_2$ | $x_3$ )

The resulting parenthesized expression is

$$( ( x_1 \mathbf{op}_1 x_2 \mathbf{op}_2 x_3 ) ) ,$$

which does not specify the evaluation order. What comes to our rescue is that C++ only allows expressions for which the magic works out! The previous bad case is impossible, for example, since all binary operators of the same precedence also have the same associativity.

Unsigned arithmetic. We have discussed how int values are converted to unsigned int values, and vice versa. The main issue (what to do with non-representable values) also occurs during evaluation of arithmetic expressions involving only one of the types. The C++ standard contains one rule for this. For all unsigned integral types, the arithmetic operators work modulo $2^b$, given b-bit representation. This means that the value of every arithmetic operation with operands of type unsigned int is well-defined. It does not necessarily give the mathematically correct value, but the unique value in the unsigned int range that is congruent to it modulo $2^b$. For example, if $a$ is a variable of type unsigned int with non-zero value, then $-a$ has value $2^b - a$.

No such rule exists for the signed integral types, meaning that over- and underflow are dealt with at the discretion of the compiler.
Sequences of + and −. We have argued above that it is usually clear which operators occur in an expression, even though some of them share their token. But since the characters + and − are heavily overused in operator tokens, special rules are needed to resolve the meanings of sequences of +’s, or of −’s.

For example, only from arities, precedences and associativities it is not clear how to interpret the expressions a+++b or −−−a. The first expression could mean (a++)+b, but it could as well mean a+(++b) or a+(+(+b)). Similarly, the second expression could either mean −(−−a), −(−a) or −(−(−a)).

The C++ standard resolves this dilemma by defining that a sequence consisting only of +’s, or only of −’s, has to be grouped into pairs from left to right, with possibly one remaining + or − at the end. Thus, a+++b means (a++)+b, and −−−a means --(−a). Note that for example the expression a++b would make sense when parenthesized as a+(+b), but according to the rule just established, it is not a well-formed expression, since a unary operator ++ cannot have operands on both sides. The expression −−−a with its logical parentheses --(−a) is invalid for another reason: the operand of the pre-increment must be an lvalue, but the expression −a is an rvalue.

Other integral types. C++ contains a number of fundamental signed and unsigned integral types. The signed ones are signed char, short int, int and long int. The standard specifies that each of them is represented by at least as many bits as the previous one in the list. The number of bits used to represent int values depends on the platform. The corresponding sequence of unsigned types is unsigned char, unsigned short int, unsigned int and unsigned long int.

These types give compilers the freedom of offering integers with larger or smaller value ranges than int and unsigned int. Smaller value ranges are useful when memory consumption is a concern, and larger ones are attractive when over- and underflow occurs. The significance of these types (which are already present in the C programming language) has faded in C++. The reason is that we can quite easily implement our own tailor-made integral types in C++, if we need them. In C this is much more cumbersome. Consequently, many C++ compilers simply make short int and long int an alias for int, and the same holds for the corresponding unsigned types.

Order of effects and sequence points Increment and decrement operators as well as assignment operators construct expressions with an effect. Such operators have to be used with care for two reasons.

The obvious reason is that (as we have already learned in the end of Section 2.1.1) the evaluation order for the sub-expressions of a given expression is not specified in general. Consequently, value and effect may depend on the evaluation order. Consider the expression

```
++i + i
```

where we suppose that i is a variable of type int. If i is initially 5, say, then the value of the composite expression may in practice be 11 or 12. The result depends on whether
or not the effect of the left operand \( ++i \) of the addition is processed before the right operand \( i \) is evaluated. The value of the expression \( ++i + i \) is therefore unspecified by the C++ standard.

To explain the second (and much less obvious, but fortunately also much less relevant) reason, let us consider the following innocent looking expression that involves a variable \( i \) of type int.

\[
i = ++i + 1
\]

This expression has two effects: the increment of \( i \) and the assignment to \( i \). Because the assignment can only happen after the operands have been evaluated, it seems that the order of the two effects is clear: the increment comes before the assignment, and the overall value and effect are well-defined.

However, this is not true, for reasons that have to do with our underlying computer model, the von Neumann architecture. From the computer's point of view, the evaluation of the sub-expression \( ++i \) consists of the following steps.

1. Copy the value of \( i \) from the main memory into one of the CPU registers;
2. Add 1 to this value in the register;
3. Write the register content back to main memory, at the address of \( i \);

Clearly, the first two steps are necessary to obtain the value of the expression \( ++i \) and, hence, have to be processed before the assignment. But the third step does not necessarily have to be completed before the assignment. In order to allow the compiler to optimize the transfer of data between CPU registers and main memory (which is very much platform dependent), this order has not been specified. In fact, it is not unreasonable to assume that the traffic between registers and main memory is organized such that several items are transferred at once or quickly after another, using so-called bursts.

Suppose as before that \( i \) initially has value 5. If the assignment is performed after the register content is written back to main memory, \( i = ++i + 1 \) sets \( i \) to 7. But if the assignment happens before, the later transfer of the register value 6 overrides the previous value of 7, and \( i \) is set to 6 instead.

The C++ standard defines a sequence point to be a point during the evaluation sequence of an expression at which is guaranteed that all effects of previously evaluated (sub)expressions have been carried out. It was probably the existence of highly optimized C compilers that let the C++ standard refrain from declaring the assignment as a sequence point. In other words, when the assignment to \( i \) takes place in the evaluation \( i = ++i + 1 \), it is not specified whether the effect of the previously evaluated increment operator has been carried out or not. In contrast, the semicolon that terminates an expression statement is always a sequence point.

Therefore, we only have an issue with expressions that have more than one effect. Hence, if you prefer not to worry about effect order, ensure that each expression that you write generates at most one effect. Expressions with more than one effect can make
sense, though, and they are ok, as long as some sequence points separate the effects and put them into a well-defined order. This is summarized in the following rule.

**Single Modification Rule**: Between two sequence points, the evaluation of an expression may modify the value of an object of fundamental type at most once.

An expression like \( i = ++i + 1 \) that violates this rule is considered semantically illegal and leads to undefined behavior.

If you perceive this example as artificial, here is a "more natural" violation of the single modification rule: if `nextvalue` is a variable of type `int`, it might seem that

\[
\texttt{nextvalue} = 5 \times \texttt{nextvalue} + 3
\]

could more compactly be written as

\[
\texttt{(nextvalue} *= 5) += 3
\]

This will compile: \((\texttt{nextvalue} *= 5)\) is an lvalue, so we can assign to it. Still, the latter expression is invalid since it modifies `nextvalue` twice.

At this point, an attentive reader should wonder how an expression that involves several output operators complies with the Single Modification Rule. Indeed, an expression like

\[
\texttt{std::cout} << a << "8 = " << b * b << ".\n"
\]

has several effects all of which modify the lvalue `std::cout`. This works since the type of `std::cout` (which we will not discuss here) is *not* fundamental and, hence, the Single Modification Rule does not apply in this case.

### 2.2.11 Goals

**Dispositional.** At this point, you should ...

1) know the three Arithmetic Evaluation Rules;
2) understand the concepts of operator precedence and associativity;
3) know the arithmetic operators for the types `int` and `unsigned int`;
4) be aware that computations involving the types `int` and `unsigned int` may deliver incorrect results, due to possible over- and underflows.

**Operational.** In particular, you should be able to ...

(G1) parenthesize and evaluate a given arithmetic expression involving operands of types `unsigned int` and `int`, the binary arithmetic operators `+`, `-`, `*`, `/`, `%`, and the unary `-` (the paragraph on parenthesizing a general expression in the Details section enables you to do this for all arithmetic operators);

(G2) derive basic statements about arithmetic expressions;
2.2. **INTEGERS**

(G3) convert a given decimal number into binary representation and vice versa;

(G4) derive the *two's complement* representation of a given number in b-bit representation, for some $b \in \mathbb{N}$;

(G5) write programs whose output is determined by a fixed number of arithmetic expressions involving literals and input variables of types *int* and *unsigned int*;

(G6) determine the value range of integral types on a given machine (using a program).

### 2.2.12 Exercises

**Exercise 13** Parenthesize the following expressions and then evaluate them step by step. This means that types and values of all intermediate results that are computed during the evaluation should be provided.

(a) $-2-4*3$  
(b) $10\%6*8\%3$  
(c) $6-3+4*5$  
(d) $5u+5*3u$  
(e) $31/4/2$  
(f) $-1-1u+1-(-1)$

**Exercise 14** Which of the following character sequences are not legal expressions, and why? (Here, a, b, and c are variables of type *int*.) For the ones that are legal, give the logical parentheses. (In order to avoid (misleading?) hints, we have removed the spaces that we usually include for the sake of better readability.)

(a) $c=a+7+++b$  
(b) $c=-a=b$  
(c) $c=a=-b$  
(d) $a-a/b*b$  
(e) $b*+++a+b$  
(f) $a-a++-b$  
(g) $7+a=b*2$  
(h) $a+3*+-b+a++$  
(i) $b++++-a$

These exercises require you to read the paragraph on logically parenthesizing a general expression in the Details section. Exercise 1) also requires you to read the paragraph on sequences of + and - in the Details section.

**Exercise 15** For all legal expressions from Exercise [14], provide a step-by-step evaluation, supposing that initially $a$ has value 5, $b$ has value 2, and the value of $c$ is undefined. Which of the expressions result in unspecified or undefined behavior?

**Exercise 16** Prove that for all $a \geq 0$ and $b, c > 0$, the following equation holds.

$$(a \div b) \div c = a \div (b\times c).$$

Does this imply that the two expressions $a/b/c$ and $a/(b\times c)$ are equivalent for all such values of the variables $a$, $b$, and $c$ (which are assumed to be of type unsigned int)?

**Exercise 17** Compute by hand binary representations of the following decimal numbers.

(a) 15  
(b) 172  
(c) 329  
(d) 1022
Exercise 18 Compute by hand decimal representations of the following binary numbers.
   (G3)
   a) 110111  b) 1000001  c) 11101001  d) 101010101

Exercise 19 By September 2009, the largest known Mersenne Prime is $2^{43,112,609} - 1$, see Section 1.1. What is the number of decimal digits that this number has? Explain how you got your answer!
   Hint: You may need the basic rules of logarithms and a pocket calculator.

Exercise 20 Assuming a 4-bit representation, compute the binary two’s complement representations of the following decimal numbers.
   (G4)
   a) 6  b) -4  c) -8  d) 9  e) -3

Exercise 21 Suppose that someone drives from A to B at an average speed of 50 km/h. On the way back from B to A, there is a traffic jam, and the average speed is only 30 km/h. What is the average speed over the whole roundtrip?
   When confronted with this question, many people would answer “40 km/h,” but this is wrong. Write a program that lets the user enter two average speeds in km/h (A → B and B → A) and computes from this the average speed over the whole roundtrip (A → B → A). Both inputs should be positive integers, and the output should be rounded down to the next smaller integer.

Exercise 22 Write a program celsius.cpp that converts temperatures from degrees Fahrenheit into degrees Celsius.
   The initial output that prompts the user to enter the temperature in degrees Fahrenheit should also contain lower and upper bounds for the allowed inputs. These bounds should be chosen such that no over- and underflows can occur in the subsequent computations, given that the user respects the bounds. You may for this exercise assume that the integer division rounds towards zero for all operands: for example, $-5 / 2$ then rounds the exact result $-2.5$ to $-2$.
   The program should output the correct result in degrees Celsius as a mixed rational number of the form $x + \frac{y}{9}$ (meaning $x + y/9$), where $x, y \in \mathbb{Z}$ and $|y| \leq 8$. For example, $13\frac{4}{9}$ could be output simply as $13 \frac{4}{9}$. We also allow for example the output $-1 -\frac{1}{9}$ (meaning $-1 - 1/9 = -10/9$).
   (G5)

Exercise 23 Write a program threebin.cpp that reads a (decimal) number $a \geq 0$ from standard input and outputs the last three bits of $a$’s binary representation. Fill up with leading zeros in case the binary representation has less than three bits. (G5)

Exercise 24 Write a program vat.cpp that computes from a net amount (in integral units of CHF) a total amount, including value-added tax (VAT). The VAT rate should be provided to the computation in form of a constant (in one tenth of a percent, to allow VAT rates like 7.6%). The net amount is the input. The output
2.2. INTEGERS

(VAT and total amount) should be rounded (down) to two digits (Rp.) after the decimal point.

An example run of the program may look like this (assuming a VAT rate of 7.6%).

Net amount = 59
VAT = 4.48
Total amount = 63.48

2.2.13 Challenges

Exercise 25 Josephus was a Jewish military leader in the Jewish-Roman war of 66-73. After the Romans had invaded his garrison town, the few soldiers (among them Josephus) that had survived the killings by the Romans decided to commit suicide. But somehow, Josephus and one of his comrades managed to surrender to the Roman forces without being killed (Josephus later became a Roman citizen and well-known historian).

This historical event is the background for the Josephus Problem that offers a (mythical) explanation about how Josephus was able to avoid suicide. Here is the problem.

41 people (numbered 0, 1, ..., 40) are standing in a circle, and every k-th person is killed until no one survives. For k = 3, the killing order is therefore 2, 5, 8, ..., 38, 0, 4, .... Where in the circle does Josephus have to position himself in order to be the last survivor (who then obviously doesn’t need to kill himself)?

a) Write a program that solves the Josephus problem; the program should receive as input the number k and output the number p(k) ∈ {0, ..., 40} of the last survivor.

b) Let us assume that Josephus is not able to chose his position in the circle, but that he can in return choose the parameter k ∈ {1, ..., 41}. Is it possible for him to survive then, no matter where he initially stands?

Hint: This exercise has a theoretical part in which you need to come up with a formula for the survivor number that you can evaluate using the subset of the C++ language that you know so far.

Exercise 26 A triple (a, b, c) of positive integers is called a Pythagorean triple if a² + b² = c². For example, (3, 4, 5) is a Pythagorean triple. Write a program pythagoras.cpp that allows you to list all Pythagorean triples for which a + b + c = 1000. We're not demanding that the program lists them directly, but the program should “prove” that your list is correct. How many such Pythagorean triples are there? (This is a slight variation of Problem 9 from the Project Euler, see http://projecteuler.net/.)
2.3 Booleans

The truth always lies somewhere else.

Folklore

This section discusses the type bool used to represent truth values or Booleans, for short. You will see a number of operations on Booleans and why only few of these operations suffice to express all the others.

You will learn how to evaluate expressions involving the type bool, using short-circuit evaluation.

What is the simplest C++ type you can think of? If we think of types in terms of their value ranges, then you will probably come up with a type whose value range is empty or consists of one possible value only. Arguably, values of such types are very easy to represent, even without spending any memory resources. However, although such types are useful in certain circumstances, you can’t do a lot of interesting computations with them. After all, there is no operation on them other than the identity.

So, let us rephrase the above question: What is the simplest non-trivial C++ type you can think of? After the above discussion we certainly have one candidate: a type with a value range that consists of exactly two elements. At first sight, such a type may again appear very limited. Nevertheless, we will see below that it allows for many interesting operations. Actually, such a type is sufficient as a basis for all kinds of computations you can imagine. (Recall, for example, that integral numbers can be represented in binary format, that is, using the two values 0 and 1 only.)

2.3.1 Boolean functions

The name “Boolean” stems from the British mathematician George Boole (1815–1864) who pioneered in establishing connections between logic and symbolic algebra. By the term Boolean function we denote a function \( f : B^n \rightarrow B \), where \( B := \{0, 1\} \) and \( n \in \mathbb{N} \). (Read 0 as false and 1 as true.)

Clearly the number of different Boolean functions is finite for every fixed \( n \); Exercise 27 asks you to show what exactly their number is. To give you a first hint: For \( n = 1 \) there are only four Boolean functions, the two constant functions \( c_0 : x \mapsto 0 \) and \( c_1 : x \mapsto 1 \), the identity \( \text{id} : x \mapsto x \) and the negation \( \text{NOT} : x \mapsto \bar{x} \), where \( \bar{0} := 1 \) and \( \bar{1} := 0 \).

In the following we restrict our focus to unary and binary Boolean functions, that is, functions from \( B \) or \( B^2 \) to \( B \). Such functions are most conveniently described as a small table that lists the function values for all possible arguments. An example for a binary Boolean function is AND : \((x, y) \mapsto x \land y\) shown in Figure 5(a). It is named AND because \( x \land y = 1 \) if and only if \( x = 1 \) and \( y = 1 \). You may guess why the
function $f : (x, y) \mapsto x \lor y$ defined in Figure 5(b) is called OR. In fact, there are two possible interpretations of the word “or”: You can read it as “at least one of”, but just as well it can mean “either ... or”, that is, “exactly one of”. The function that corresponds to the latter interpretation is shown in Figure 5(c). It is usually referred to as XOR : $(x, y) \mapsto x \oplus y$ or exclusive or. Figure 5(e) depicts the table for the unary function NOT.

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x \land y$</th>
<th>$x \lor y$</th>
<th>$x \oplus y$</th>
<th>$x \uparrow y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(a) AND. (b) OR. (c) XOR. (d) NAND. (e) NOT.

Figure 4: Examples for Boolean functions.

Completeness. Figure 4 shows just a few examples. However, in a certain sense, it shows you everything about binary Boolean functions. Some of these functions are so fundamental that every binary Boolean function can be generated from them. For example, XOR can be generated from AND, OR and NOT:

$$\text{XOR}(x, y) = \text{AND} (\text{OR}(x, y), \text{NOT} (\text{AND}(x, y))).$$

Informally, “either or” means “or” but not “and”. Formulas like this are easily checked by going through all (four) possible combinations of arguments.

Similarly, the function NAND : $(x, y) \mapsto x \uparrow y$ described in Figure 5(d) can be generated from NOT and AND (hence the name):

$$\text{NAND}(x, y) = \text{NOT} (\text{AND}(x, y)).$$

Let us define what we mean by “generate”.

**Definition 2** Consider a set $F$ of boolean functions. A binary boolean function $f$ is called **generated** by $F$ if $f$ can be expressed by a formula that only contains the variables $x$ and $y$, the constants 0 and 1, and the functions from $F$.

For a set $F$ of binary functions, a set $F$ of binary functions is said to be **complete** if and only if every function $f \in F$ can be generated by $F$.

We are now prepared for a completeness proof.

**Theorem 1** The set of functions $\{\text{AND}, \text{OR}, \text{NOT}\}$ is complete for the set of binary Boolean functions.
Proof. Every binary Boolean function \( f \) is completely described by its characteristic vector \((f(0,0), f(0,1), f(1,0), f(1,1))\). For example, AND has characteristic vector \((0,0,0,1)\), or 0001 for short. Let \( f_{b_1b_2b_3b_4} \) denote the Boolean function with characteristic vector \( b_1b_2b_3b_4 \). For example, AND \( = f_{0001} \).

In the first step of the proof, we show that all those functions can be generated whose characteristic vector contains a single 1. Indeed,

\[
\begin{align*}
  f_{0001}(x, y) &= \text{AND}(x, y), \\
  f_{0010}(x, y) &= \text{AND}(x, \text{NOT}(y)), \\
  f_{0100}(x, y) &= \text{AND}(y, \text{NOT}(x)), \\
  f_{1000}(x, y) &= \text{NOT}(\text{OR}(x, y)).
\end{align*}
\]

To check the formula for \( f_{0010} \), for example, we can create a table for the function \( \text{AND}(x, \text{NOT}(y)) \) as in Figure 4 and convince ourselves that the resulting characteristic vector is 0010.

In the second step, we show that every function whose characteristic vector is nonzero can be generated. This is done by combining the already generated “single-1” functions through OR, which simply adds up their 1’s. For example,

\[
\begin{align*}
  f_{1100}(x, y) &= \text{OR}(f_{1000}(x, y), f_{0100}(x, y)), \\
  f_{0111}(x, y) &= \text{OR}(\text{OR}(f_{0100}(x, y), f_{0010}(x, y)), f_{0001}(x, y)).
\end{align*}
\]

We abstain from working this argument out formally, since we believe that you get its idea. Finally, we generate \( f_{0000} \) as

\[
f_{0000}(x, y) = 0.
\]

Exercise 30 asks you to show that the sets \( \{\text{AND, NOT}\} \), \( \{\text{OR, NOT}\} \), and even the set that consists of the single function \( \{\text{NAND}\} \) are complete for the set of binary Boolean functions.

2.3.2 The type bool

In C++, Booleans are represented by the fundamental type bool. Its value range consists of the two elements \texttt{true} and \texttt{false} that are associated with the literals \texttt{true} and \texttt{false}, respectively. For example,

```cpp
bool b = true;
```

defines a variable \( b \) of type bool and initializes it to \texttt{true}.

Formally, the type \texttt{bool} is an integral type, defined to be less general than \texttt{int} (which in turn is less general than \texttt{unsigned int}, see Section 2.2.7). An expression of type bool, or of a type whose values can be converted to bool, is called a \textit{predicate}.
2.3. **BOOLEANS**

**Logical operators.** The complete set of binary Boolean functions is available via the *logical operators* `&&` (AND), `||` (OR), and `!` (NOT). Compared to the notation used in Section [2.3.1](#) we simply identify 1 with *true* and 0 with *false*. Both `&&` and `||` are binary operators, while `!` is unary. All operands are rvalues of type `bool`, and all logical operators also return rvalues of type `bool`. Like in logics, `&&` binds more strongly than `||`, and `!` binds more strongly than `&&` (recall that an operator binds more strongly than another if its has higher precedence).

**Relational operators.** There is also a number of operators on arithmetic types whose result is of type `bool`. For each arithmetic type there exist the six *relational operators* `<, >, <=, >=, ==, and !=`. These are binary operators whose two rvalue operands are of some arithmetic type and whose result is an rvalue of type `bool`. The operators `<=` and `>=` correspond to the mathematical relations `≤` and `≥`, respectively. The operator `==` tests for equality and `!=` tests for inequality.

Since `bool` is an integral type, the relational operators may also have operands of type `bool`. The respective comparisons are done according to the convention `false<true`.

Watch out! A frequent beginner’s mistake is to use the assignment operator `=` where the equality operator `==` is meant.

As a general rule, arithmetic operators bind more strongly than relational ones, and these in turn bind more strongly than the logical operators.

**Boolean Evaluation Rule:** Binary arithmetic operators have higher precedence than relational operators, and these have higher precedence than binary logical operators.

All binary relational and logical operators are (as the binary arithmetic operators) left-associative, see also Table [2](#). For example, the expression

```
7 + x < y && y != 3 * z
```

is logically parenthesized as

```
((7 + x) < y) && (y != (3 * z)).
```

Be careful with mathematical shortcut notation such as `a = b = c`. As a C++ expression,

```
a == b == c
```

is not equivalent to

```
a == b && b == c.
```

By left associativity of `==`, the expression `a == b == c` is logically parenthesized as

```
(a == b) == c.
```

If all of `a`, `b`, and `c` are variables of type `int` with value 0, the evaluation yields

```
(0 == 0) == 0 → true == 0 → 1 == 0 → false,
```

just the opposite of what you usually mean by `a = b = c`. 

De Morgan's laws. The well-known formulae of how to express AND in terms of OR and vice versa with the help of NOT, are named after the British mathematician Augustus De Morgan (1806–1871). He was a pioneer in symbolic algebra and logics. Also the rigorous formulation of “mathematical induction” as we know and use it today goes back to him. The de-Morgan-formulae state that (in C++-language)

\[ !(x \&\& y) == (!x || !y) \]

and

\[ !(x || y) == (!x \&\& !y) \].

These formulae can often be used to transform a Boolean expression (an expression of type \texttt{bool}) into a “simpler” equivalent form. For example,

\[ !(x < y || x + 1 > z) \&\& !(y <= 5 * z || !(y > 7 * z)) \]

can equivalently be written as

\[ x >= y \&\& x + 1 <= z \&\& y > 5 * z \&\& y > 7 * z \]

which is clearly preferable in terms of readability.

For more details about precedences and associativities of the logical and relational operators, see Table 2. You may find this information helpful in order to solve Exercise 32.

<table>
<thead>
<tr>
<th>Description</th>
<th>Operator</th>
<th>Arity</th>
<th>Prec.</th>
<th>Assoc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical not</td>
<td>!</td>
<td>1</td>
<td>16</td>
<td>right</td>
</tr>
<tr>
<td>less</td>
<td>&lt;</td>
<td>2</td>
<td>11</td>
<td>left</td>
</tr>
<tr>
<td>greater</td>
<td>&gt;</td>
<td>2</td>
<td>11</td>
<td>left</td>
</tr>
<tr>
<td>less or equal</td>
<td>&lt;=</td>
<td>2</td>
<td>11</td>
<td>left</td>
</tr>
<tr>
<td>greater or equal</td>
<td>&gt;=</td>
<td>2</td>
<td>11</td>
<td>left</td>
</tr>
<tr>
<td>equality</td>
<td>==</td>
<td>2</td>
<td>10</td>
<td>left</td>
</tr>
<tr>
<td>inequality</td>
<td>!=</td>
<td>2</td>
<td>10</td>
<td>left</td>
</tr>
<tr>
<td>logical and</td>
<td>&amp;&amp;</td>
<td>2</td>
<td>6</td>
<td>left</td>
</tr>
<tr>
<td>logical or</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

Table 2: Precedences and associativities of logical and relational operators. All operands and return values are rvales.

Conversion and promotion. It is possible that the two operands of a relational operator have different type. This case is treated in the same way as for the arithmetic operators. The composite expression is evaluated on the more general type, to which the operand of the less general type is implicitly converted. In particular, \texttt{bool} operands are converted to the respective integral type of the other operand. Here, the value \texttt{false} is converted to 0, and \texttt{true} to 1. If the integral type is \texttt{int}, this conversion is defined to be a promotion. A promotion is a special conversion for which the C++ standard guarantees that no information gets lost.
2.3. BOOLEANS

The conversion goes into the other direction for logical operators. In mixed expressions, the integral operands of logical operators are converted to \texttt{bool} in such a way that \texttt{0} is converted to \texttt{false} and any other value is converted to \texttt{true}.

These conversions also take place in initializations and assignments, as in the following examples.

\begin{verbatim}
bool b = 5;  // b is initialized to true
int i = b;  // i is initialized to 1
\end{verbatim}

2.3.3 Short circuit evaluation

The evaluation of expressions involving logical and relational operators proceeds according to the general rules, as discussed in Sections 2.2.1 and 2.2.3. However, there is one important difference regarding the order in which the operands of an operator are evaluated. While in general this order is unspecified, the binary logical operators \&\& and || always guarantee that their left operand is evaluated first. Moreover, if the value of the composite expression is already determined by the value of the left operand then the right operand is \textit{not evaluated} at all. This evaluation scheme is known as \textit{short circuit evaluation}.

How can it happen that the final value is already determined by the left operand only? Suppose that in an \&\& operator the left operand evaluates to \texttt{false}; then no matter what the right operand gives, the result will always be \texttt{false}. Hence, there is no need to evaluate the right operand at all. The analogous situation occurs if in an || operator the left operand evaluates to \texttt{true}.

At first sight it looks as if short circuit evaluation is merely a matter of efficiency. But there is another benefit. It occurs when dealing with expressions that are defined for certain parameters only. Consider for example the division operation that is defined for a nonzero divisor only. Due to short circuit evaluation, we can write

\begin{verbatim}
x != 0 && z / x > y
\end{verbatim}

and be sure that this expression is always valid. If the right operand was evaluated for \texttt{x} having value \texttt{0}, then the result would be undefined.

2.3.4 Details

\textbf{Naming.} The XOR function is also frequently called \textit{antivalence} and denoted by $\leftrightarrow$. The NAND function is also known as \textit{alternate denial} or \textit{Sheffer stroke}. The latter name is after the American mathematician Henry M. Sheffer (1883–1964) who proved that all other logical operations can be expressed in terms of NAND.

\textbf{Bitwise operators.} We have seen in Section 2.2.8 that integers can be represented in binary format, that is, as a sequence of bits each of which is either 0 or 1. Boolean
functions can naturally be extended to integral types by applying them bitwise to the binary representations.

**Definition 3** Consider a nonnegative integer $b$ and two integers $x = \sum_{i=0}^{b} a_i 2^i$ and $y = \sum_{i=0}^{b} b_i 2^i$, for which $a_i, b_i \in \{0, 1\}$ for all $0 \leq i \leq b$.

For a unary Boolean function $f : \{0, 1\} \to \{0, 1\}$ the bitwise operator $\varphi_f$ corresponding to $f$ is defined as $\varphi_f(x) = \sum_{i=0}^{b} f(a_i) 2^i$.

For a binary Boolean function $g : \{0, 1\}^2 \to \{0, 1\}$ the bitwise operator $\varphi_g$ corresponding to $g$ is defined as $\varphi_g(x, y) = \sum_{i=0}^{b} g(a_i, b_i) 2^i$.

For illustration, suppose that we have an unsigned integral type with a 4-bit representation. That is, 0000 represents 0, 0001 represents 1, and so on, up to 1111 which represents 15.

Then you can check that $\varphi_{\text{OR}}(4, 13) = 13$, $\varphi_{\text{NAND}}(13, 9) = 6$, and $\varphi_{\text{NOT}}(2) = 13$.

Several bitwise operators are defined for the integral types in C++. There is a bitwise AND $\&$, a bitwise OR $\mid$, and a bitwise XOR $\sim$, as well as a bitwise NOT $\sim$ that is usually referred to as complement. As the arithmetic operators, the binary bitwise operators (except for $\sim$) have a corresponding assignment operator. The precedences and associativity of these operators are listed in Table 3.

<table>
<thead>
<tr>
<th>Description</th>
<th>Operator</th>
<th>Arity</th>
<th>Prec.</th>
<th>Assoc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitwise complement</td>
<td>$\sim$</td>
<td>1</td>
<td>16</td>
<td>right</td>
</tr>
<tr>
<td>bitwise and</td>
<td>$&amp;$</td>
<td>2</td>
<td>9</td>
<td>left</td>
</tr>
<tr>
<td>bitwise xor</td>
<td>$\sim$</td>
<td>2</td>
<td>8</td>
<td>left</td>
</tr>
<tr>
<td>bitwise or</td>
<td>$\mid$</td>
<td>2</td>
<td>7</td>
<td>left</td>
</tr>
<tr>
<td>and assignment</td>
<td>$&amp;=$</td>
<td>2</td>
<td>4</td>
<td>right</td>
</tr>
<tr>
<td>xor assignment</td>
<td>$\sim=$</td>
<td>2</td>
<td>4</td>
<td>right</td>
</tr>
<tr>
<td>or assignment</td>
<td>$\mid=$</td>
<td>2</td>
<td>4</td>
<td>right</td>
</tr>
</tbody>
</table>

**Table 3:** Precedence and associativity of bitwise operators.

Note that the functionality of these operators is implementation defined, since the bitwise representations of integral type values are not specified by the C++ standard. We have only discussed the most frequent (and most likely) such representations in Section 2.2.8. You should therefore only use these operators when you know the representation. Even then, expressions involving the bitwise operators are implementation defined.

This is most obvious with the bitwise complement: even if we assume the standard binary representation of Section 2.2.8 the value of the expression $\sim 0$ depends on the number $b$ of bits in the representation. This value therefore changes when you switch from a 32-bit machine to a 64-bit machine.
2.3. BOOLEANS

2.3.5 Goals

Dispositional. At this point, you should ...

1) know the basic terminology around Boolean functions and understand the concept of completeness;
2) know the type bool, its value range, and the conversions and operations involving bool;
3) understand the evaluation of expressions involving logical and relational operators, in particular the Boolean Evaluation Rule and the concept of short circuit evaluation.

Operational. In particular, you should be able to ...

(G1) prove or disprove basic statements about Boolean functions;
(G2) prove whether or not a given set of binary Boolean functions is complete;
(G3) evaluate a given expression involving arithmetic, logical, and relational operators;
(G4) read and understand a given simple program (see below), involving objects of arithmetic type (including bool) and arithmetic, logical, and relational operators.

The term simple program refers to a program that consists of a main function which in turn consists of a sequence of declaration and expression statements. Naturally, only the fundamental types and operations discussed in the preceding sections are used.

2.3.6 Exercises

Exercise 27 For \( n \in \mathbb{N} \), how many different Boolean functions \( f : \mathbb{B}^n \rightarrow \mathbb{B} \) exist? (G1)

Exercise 28 Prove or disprove that for all \( x, y, z \in \mathbb{B} \) (G1)

\[
\begin{align*}
\text{a)} & \quad (x \oplus y) \oplus z = x \oplus (y \oplus z). \quad \text{(i.e., XOR is associative)} \\
\text{b)} & \quad (x \land y) \lor z = (x \lor y) \land (y \lor z). \quad \text{(i.e., (AND, OR) is distributive)} \\
\text{c)} & \quad (x \lor y) \land z = (x \land z) \lor (y \land z). \quad \text{(i.e., (OR, AND) is distributive)} \\
\text{d)} & \quad (x \uparrow y) \uparrow z = x \uparrow (y \uparrow z). \quad \text{(i.e., NAND is associative)}
\end{align*}
\]

Exercise 29 For \( x_1, \ldots, x_n, n \in \mathbb{N} \), give a verbal description of \( x_1 \oplus x_2 \oplus \ldots \oplus x_n \) in terms of the \( x_i, 1 \leq i \leq n \). (G1)

Exercise 30 Show that the following sets of functions are complete for the set of binary Boolean functions. (G2)

\[
\begin{align*}
\text{a)} & \quad \{\text{AND, NOT}\}
\end{align*}
\]
b) \{\text{OR}, \text{NOT}\}

c) \{\text{NAND}\}

d) \{\text{NOR}\}, \text{ where } \text{NOR} := \text{NOT } \circ \text{OR}.

e) \{\text{XOR}, \text{AND}\}

You may use the fact that set \{\text{AND}, \text{OR}, \text{NOT}\} is complete for the set of binary Boolean functions.

**Exercise 31** Suppose \(a, b, \) and \(c\) are all variables of type \text{int}. Find values for \(a, b, \) and \(c\) for which the expressions \(a < b < c\) and \(a < b && b < c\) yield different results.

**Exercise 32** Parenthesize the following expressions according to operator precedences and associativities.

a) \(x != 3 < 2 || y && -3 <= 4 - 2 * 3\)

b) \(z > 1 && ! x != 2 - 2 == 1 && y\)

c) \(3 * z > z || 1 / x != 0 && 3 + 4 >= 7\)

**Exercise 33** Evaluate the expressions given in Exercise 32 step-by-step, assuming that \(x, y, \) and \(z\) are all of type \text{int} with \(x==0, y==1, \) and \(z==2\).

**Exercise 34** What can you say about the output of the following program? Characterize it depending on the input and explain your reasoning.

```cpp
1 #include <iostream>
2 int main()
3 {
4     int a;
5     std::cin >> a;
6     std::cout << (a++ < 3) << ".\n";
7     const bool b = a * 3 > a + 4 && !(a >= 5);
8     std::cout << (!b || ++a > 4) << ".\n";
9     return 0;
10 }
```

**Exercise 35** Find the logical parentheses in lines 9 and 12 of the following program. What can you say about the output of the following program? Characterize it depending on the input and explain your reasoning.
2.3. BOOLEANS

```cpp
#include <iostream>

int main ()
{
    unsigned int a;
    std::cin >> a;

    unsigned int b = a;
    b /= 2 + b / 2;
    std::cout << b << "\n";

    const bool c = a < 1 || b != 0 && 2 * a / (a - 1) > 2;
    std::cout << c << "\n";
    return 0;
}
```

2.3.7 Challenges

Exercise 36 The Reverse Polish Notation (RPN) is a format of writing expressions without any parentheses. RPN became popular in the late nineteen-sixties when the company Hewlett-Packard started to use it as input format for expressions on their desktop and handheld calculators.

In RPN, we first write the operands, and then the operator (that’s what the Reverse stands for). For example, the expression

```
AND(OR(0, NOT(AND(0, 1))), 1)
```

can be written like this in RPN:

```
0 0 1 AND NOT OR 1 AND.
```

The latter sequence of operands and operators defines a specific evaluation sequence of the expression, see Section 2.2.3. To evaluate an expression in RPN, we go through the sequence from left to right; whenever we find an operand, we don’t do anything, but when we find an operator (of arity n), we evaluate it for the n operands directly to the left of it and replace the involved n+1 sequence elements by the result of the evaluation. Then we go to the next sequence element. In case of our example above, this proceeds as follows (currently processed operator in bold):
0 0 1 \textbf{AND} NOT OR 1 \textbf{AND} \\
0 0 \textbf{NOT} OR 1 \textbf{AND} \\
0 1 \textbf{OR} 1 \textbf{AND} \\
1 1 \textbf{AND} \\
1

To see that this is indeed a way of evaluating the original expression

\begin{align*}
\text{AND}(\text{OR}(0, \text{NOT}(\text{AND}(0, 1))), 1),
\end{align*}

you can for example make a bottom-up drawing of an expression tree (Section \ref{sec:expression-trees}) that corresponds to the evaluation sequence in RPN. You will find that this tree is also valid for the original expression.

Here comes the actual exercise. Write programs \texttt{and.cpp}, \texttt{or.cpp}, and \texttt{not.cpp} that receive as input a sequence \texttt{s} of boolean values in \{0, 1\} ("all operands to the left of the operator"). The output should be the sequence \texttt{s'} that we get by replacing the last \( n \) operands in \texttt{s} with the result of evaluating the respective operator for them. In case of \texttt{and.cpp} and \texttt{or.cpp}, we use \( n = 2 \), and for \texttt{not.cpp} \( n = 1 \). For example, on input \((1, 1, 0)\), program \texttt{and} should output the sequence \((1, 0)\), while \texttt{not} should yield \((1, 1, 1)\).

In addition, write programs \texttt{zero.cpp} and \texttt{one.cpp} that output the sequence \texttt{s'} obtained by appending a 0 or 1 to the input \texttt{s}. Finally, write a program \texttt{eval.cpp} (with no input) that outputs the empty sequence.

The goal of all this is to evaluate boolean functions in RPN by simply calling the corresponding sequence of programs (preceded by a call to \texttt{eval}), where the output of one program is used as input for the next one in the sequence. In Unix and Linux this can elegantly be done via a pipe. For example, to evaluate the example expression from above in RPN, we simply type the command

```
./eval | ./zero | ./zero | ./one | ./and | ./not | ./or | ./one | ./and
```

This calls all listed programs in turn, where a separating pipe symbol | has the effect that the output of the program to the left of it is used as the input for ("is piped into") the program to the right of it.

Consequently, the whole aforementioned command should simply write 1 to standard output, the result of the evaluation. Also test your programs with some other RPN sequences, in particular the "obvious" ones of the form

```
./eval | ./zero | ./one | ./or
```
(this one should output 1) to make sure that they work as expected.

**Hint:** It is not necessary that your programs accept sequences $s$ of arbitrary length as input. A maximum length of 32, for example, is sufficient for all practical purposes; in this case, the sequence can be encoded by one value of type unsigned int.
2.4 Control statements

We are what we repeatedly do. Excellence, then, is not an act but a habit.


This section introduces four concepts to control the execution of a program: selection, iteration, blocks, and jumps. These concepts enable us to deviate from the default linear control flow which executes statement by statement from top to bottom. You will learn how these concepts are implemented in C++, and how to apply them to create interesting programs.

The programs that we have seen so far are all pretty simple. They consist of a sequence of statements that are executed one by one from the first to the last. Such a program is said to have a linear control flow. This type of control flow is quite restrictive, as each statement in the source code is executed at most once during the execution of the program. Suppose you want to implement an algorithm that performs 10,000 steps for some input. Then you would have to write a program with at least 10,000 lines of code. Obviously this is undesirable. Therefore, in order to implement non-trivial algorithms, more powerful mechanisms to control the flow of a program are needed.

2.4.1 Selection: if– and if-else statements

One particularly simple way to deviate from linear control flow is to select whether or not a particular statement is executed. In C++ this can be done via an if statement. The syntax is

```cpp
if ( condition )
    statement
```

where condition is an expression or variable declaration of a type whose values can be converted to bool, and statement — as the name suggests — is a statement. In case you are missing a semicolon after statement: recall that this semicolon is part of the statement. The semantics is the following: condition is evaluated; if and only if its value is true, statement is executed afterwards. In other words, an if statement splits the control flow into two branches. The value of condition selects which of these branches is executed. For example, the following lines of code
2.4. CONTROL STATEMENTS

int a;
std::cin >> a;
if (a % 2 == 0) std::cout << "even";

read a number from standard input into the variable a and write “even” to standard output if and only if a is even.

Optionally, an if statement can be complemented by an else-branch. The syntax is

```
if ( condition )
   statement1
else
   statement2
```

and the semantics is as follows: condition is evaluated; if its value is true, statement1 is executed afterwards; otherwise, statement2 is executed afterwards. For example, the following lines of code

```
int a;
std::cin >> a;
if (a % 2 == 0)
   std::cout << "even";
else
   std::cout << "odd";
```

read a number from standard input into the variable a. Then if a is even, “even” is written to standard output; otherwise, “odd” is written to standard output.

When formatting an if statement, it is common to insert a line break before statement1, before else, and before statement2. Moreover, statement1 and statement2 are indented and else is aligned with if, as shown in the example above. If the whole statement fits on a single line then it can also be typeset as a single line.

Collectively, if- and if-else statements are known as selection statements.

2.4.2 Iteration: for statements

A much more powerful way of manipulating the control flow is provided by iteration statements. Iteration allows to execute a statement many times, possibly with different parameters each time. Iteration statements are also called loops, as they “loop through” a statement (potentially) several times. Selection and iteration statements are collectively referred to as control statements.

Consider the problem of computing the sum $S_n = \sum_{i=1}^{n} i$ of the first n natural numbers, for a given $n \in \mathbb{N}$. Program reads in a variable n from standard input, defines another variable s to contain the result, computes the result and finally outputs it. In order to understand why the program sum_n.cpp indeed behaves as claimed, we have to explain the different parts of a for statement.
for statement. The for statement is a very compact form of an iteration statement, as it combines three statements or expressions into one. In most cases, the for statement serves as a “counting loop” as in Program 7. Its syntax is defined by

```
for ( init-statement condition; expression )
  statement
```

where init-statement is an expression statement, a declaration statement, or the null statement, see Section 2.1.15. In all of these cases, init-statement ends with a semicolon in between the parentheses after a for. Usually init-statement defines and initializes a variable that is used to control and eventually end the iteration statement’s execution. In sum_n.cpp, init-statement is a declaration statement that defines the variable i.

As in an if statement, condition is an expression or variable declaration whose type can be converted to bool. It defines how long the iteration goes on, namely as long as condition returns true. It is allowed that condition is empty in which case its value is interpreted as true. As the name suggests, expression is an arbitrary expression that may also be empty (in which case it has no effect). statement is an arbitrary statement. It is referred to as the body of the for statement.
Typically, expression has the effect of changing a value that appears in condition. Such an effect is said to "make progress towards termination". The goal is that condition is false after expression has been evaluated a finite number of times. In that sense, every evaluation of expression makes a step towards the end of the for statement. In sum_n.cpp, expression increments the variable i which is bounded from above in condition. In cases like this where the value of a single variable is accessed and changed by condition and expression, we call this variable the control variable of the for statement.

We are now ready to precisely define the semantics of a for statement. First, init-statement is executed once. Thereafter condition is evaluated. If it returns true, an iteration of the loop starts. If condition returns false, the for statement terminates, that is, its processing ends immediately.

Each single iteration of a for statement consists of first executing statement and then evaluating expression. After each iteration, condition is evaluated again. If it returns true, another iteration follows. If condition returns false, the for statement terminates. The execution order is therefore init-statement, condition, statement, expression, condition, statement, expression, ... until condition returns false.

Let's see this in action: Consider the for statement

\[
\text{for (unsigned int i = 1; i <= n; ++i) s += i;}
\]

in sum_n.cpp and suppose n == 2. First, the variable i is defined and initialized to 1. Then it is tested whether i <= n. As 1 <= 2 is true, the first iteration starts. The statement s += i is executed, setting s to 1, and thereafter i is incremented by one such that i == 2. One iteration is now complete. As a next step, the condition i <= n is evaluated again. As 2 <= 2 is true, another iteration follows. First s += i is executed, setting s to 3. Thereafter, i is incremented by one such that i == 3. The second iteration is now complete. The subsequent evaluation of i <= n entails 3 <= 2 which is false. Thus, no further iteration takes place and the processing of the for statement ends. The value of s is now 3, the sum of the first 2 natural numbers.

Infinite loops. It is easily possible to create loops that do not terminate. For example, recall that both condition and expression may be empty. Moreover, both init-statement and statement can be the null statement. In this case we get the for statement

\[
\text{for ( ; ; ) ;}
\]

As the empty condition has value true, executing this statement runs through iteration after iteration without actually doing anything. Therefore, for ( ; ; ) may be read as "forever". In general, a statement which does not terminate is called an infinite loop.

Clearly, infinite loops are extremely undesirable and programmers try hard to avoid them. Nevertheless, sometimes such loops occur even in real life software. If you regularly use a computer, you have probably experienced this kind of phenomenon: a program "hangs".
You may ask: Why doesn’t the compiler simply detect infinite loops and warns me about them just as it complains about syntax errors? Indeed, this would be a great thing to have and it would solve many problems in software development. The problem is that infinite loops are not always as easy to spot as in the above example. Loops can be pretty complicated, and possibly they loop infinitely when executed in certain program states only.

In fact, the situation is hopeless: It can be shown that the problem of detecting infinite loops (commonly referred to as the halting problem) cannot be solved by a computer, as we have and understand it today (see the Details). Therefore, some care is needed when designing loops. We have to check “by hand” that the iteration statement terminates for all possible program states that can occur.

Gauss. You may know or have realized that our program sum_n.cpp is actually a bad example. It is bad in the sense that it does not convincingly demonstrate the power of control statements.

In his primary school days, the German mathematician Carl Friedrich Gauss (1777–1855) was told to sum up the numbers 1, 2, 3, . . . , 100. The teacher had planned to keep his students busy for a while, but Gauss came up with the correct result 5050 very quickly. He had imagined writing down the numbers in increasing order, and one line below once again in decreasing order. Clearly, the two numbers in each column sum up to 101; hence, the overall sum is 100 · 101 = 10100, half of which is the number that was asked for.

\[
\begin{array}{cccccccc}
1 & 2 & 3 & \ldots & 98 & 99 & 100 \\
100 & 99 & 98 & \ldots & 3 & 2 & 1 \\
101 & 101 & 101 & \ldots & 101 & 101 & 101 \\
\end{array}
\]

In this way, Gauss discovered the formula

\[\sum_{i=1}^{n} i = n(n+1)/2,\]

for every \( n \in \mathbb{N} \). The for statement in sum_n.cpp can therefore be replaced by the much more elegant and efficient statement

\[s = n \times (n + 1) / 2;\]

Note that in this statement, the integer division coincides with the real division, since for all \( n \), the product \( n(n+1) \) is even. We next get to a real application of selection and iteration statements.

Prime numbers. In the introductory Section 1.1 we have talked a lot about prime numbers. How would a program look like that tests whether or not a given number is prime? A number \( n \in \mathbb{N}, n \geq 2 \) is prime if and only if it is not divisible by any number \( d \in \{2, \ldots, n - 1\} \). The strategy for our program is therefore clear: Write a loop that
runs through all these numbers, and test each of them for being a divisor of \( n \). If a divisor is found, we can stop and output a factorization of \( n \) into two numbers, proving that \( n \) is not prime. Otherwise, we output that \( n \) is prime. Program 8 implements this strategy in C++, using one for statement, and one if statement. Remarkably, the for statement has an empty body, since we have put the divisibility test into the condition. The important observation is that the condition \( n \% d \neq 0 \) definitely returns false for \( d = n \), so that the loop is guaranteed to terminate; if (and only if) condition returns false earlier, we have found a divisor of \( n \) in the range \( \{2, \ldots, n-1\} \).

```cpp
1 // Program: prime.cpp
2 // Test if a given natural number is prime.
3 #include <iostream>
4
5 int main ()
6 {
7     // Input
8     unsigned int n;
9     std::cout << "Test if n>1 is prime for n =? ";
10    std::cin >> n;
11
12     // Computation: test possible divisors d
13     unsigned int d;
14     for (d = 2; n % d != 0; ++d);
15
16     // Output
17     if (d < n)
18         // d is a divisor of n in \{2,\ldots,n-1\}
19         std::cout << n << " = " << d << " * " << n / d << ".
20     else
21         // no proper divisor found
22         std::cout << n << " is prime.\n";
23
24     return 0;
25 }
```

Program 8: ../progs/lecture/prime.cpp

We would like to point out that the above arguments are only valid for \( n \geq 2 \); Exercise 45 asks you to consider the cases \( n = 0, 1 \).

### 2.4.3 Blocks and scope

In C++ it is possible to group a sequence of one or more statements into one single statement that is then called a compound statement, or simply a block. This mechanism
does not manipulate the control flow directly. Blocks allow to structure a program by grouping statements that logically belong together. In particular, they are a tool to design powerful and at the same time readable control statements.

Syntactically, a block is simply a sequence of zero or more statements that are enclosed in curly braces.

```
{ statement1 statement2 ... statementN }
```

Each of the statements may in particular be a block, so it is possible to have nested blocks. The simplest block is the empty block `{}`.

You have already seen blocks. Each program contains a special block, the so-called function body of the main function. This block encloses the sequence of statements that is executed when the main function is called by the operating system.

Using blocks, one can create selection and iteration statements whose body contains a sequence of two or more statements. For example, suppose that for testing purposes we would like to write out all partial sums during the computation in `sum_n.cpp`:

```cpp
for (unsigned int i = 1; i <= n; ++i) {
    s += i;
    std::cerr << i << "-th partial sum is " << s << "\n";
}
```

Here two statements are executed in each iteration of the loop. First, the next summand is added to `s`, then the current value of `s` is written to standard error output.

Blocks should in general be formatted as shown above. That is, a line break appears after the opening and before the closing brace, and all lines in between are indented one level. Only if the block consists of just one single statement and it all fits on one line, the block can be formatted as one single line.

The kind of test output we have created in the previous example is called debugging output. A bug is a commonly used term to denote a programming error, hence “debugging” is the process of finding and eliminating such errors. It is good practice to write debugging output to standard error output since it can then more easily be separated from the “real” program output that usually goes to standard output.

Visibility. Blocks do not only structure a program visually but they also provide a logical boundary around declarations (of variables, for example). Every declaration that appears inside a block is called local to that block. A local declaration extends only until the end of the block in which it appears. A name that is introduced by a local declaration is not “visible” outside of the block where it is declared. For example, in

```cpp
int main() {
    { 
        const int i = 2;
    }
```
2.4. CONTROL STATEMENTS

```cpp
6   std::cout << i; // error, undeclared identifier
7   return 0;
8 }
```

the variable `i` declared inside the block in line 3–5 is not visible in the output statement in line 6. Thus, if you confront the compiler with this code, it issues an error message.

Control statements and blocks. Control statements act like blocks themselves. Therefore every declaration appearing in a control statement is local to that control statement. In particular, this applies to a variable defined in the `init-statement` of a `for` statement. For example, in

```cpp
1   int main()
2 {
3     for (unsigned int i = 0; i < 10; ++i) s += i;
4     std::cout << i; // error, undeclared identifier
5     return 0;
6 }
```

the expression `i` in line 4 does not refer to the variable `i` defined in line 3.

Declarative region. After having seen these first examples, we will now introduce the precise terminology that allows us to deduce which names can be used where in the program. Each declaration has an associated declarative region. This region is the part of the program in which the declaration appears. Such a region can be a block, a function definition, or a control statement. In all these cases the declaration is said to have local scope. A declaration can also have namespace scope, if it appears inside a namespace, see Section 2.1.3. Finally, a declaration that is outside of any particular other structure has global scope.

Scope. A name introduced by a declaration D is valid or visible in a part of its declaration’s declarative region, called the scope of the declaration. Within the scope of D, the name introduced by D may be used and actually refers to the declaration D. In most cases, the scope of a declaration is equal to its potential scope.

The potential scope of a declaration starts at the point where the declaration appears. For the name to be declared this is called its point of declaration. The potential scope extends until the end of the declarative region.

To get the scope of a declaration, we start from its potential scope but we possibly have to remove some parts of it. This happens when the potential scope contains one or more declarations of the same name. As an example, consider Program 9.

```cpp
1 #include <iostream>
2
3 int main()
```


```cpp
4 { 
5   const int i = 2;
6   for (int i = 0; i < 5; ++i)
7   std::cout << i; // outputs 0, 1, 2, 3, 4
8   std::cout << i; // outputs 2
9   return 0;
10 }
```

Program 9: `../progs/lecture/scope.cpp`

The `i` in line 7 refers to the declaration from line 6, whereas the `i` in line 8 refers to the declaration from line 5. Therefore, the program outputs first 0, 1, 2, 3, 4, and then 2. In some sense, the declaration in line 6 temporarily hides the previous declaration of `i` from line 5. This phenomenon is called *name hiding*. But when the declarative region of the second declaration ends in line 7, the second declaration “becomes invisible” (we say: “it runs out of scope”) and the first declaration takes over again. In particular, since the name `i` in line 8 refers to the constant defined in line 5, we get the output 2 in line 8.

It is good practice to avoid name hiding since this unnecessarily obfuscates the program. On the other hand, name hiding allows us (like in Program 9) to use our favorite identifier `i` as the name of the control variable in a for statement, without having to check whether there is some other name `i` somewhere else in the program. This is an acceptable and even useful application of name hiding.

Now we can get to the formal definition of scope in the general case (possible presence of multiple declarations of the same name). The *scope* of a declaration `D` is obtained from its potential scope as follows: For each declaration `E` in the potential scope of `D` such that both `D` and `E` declare the same name, the potential scope of `E` is removed from the scope of `D`. Figure 5 gives a symbolic picture of the situation.

In Program 9 the declarative region of the declaration in line 5 is line 4–10 (a block), its potential scope is line 5–10, and its scope is line 5 plus line 8–10. For the declaration in line 6, declarative region (a control statement), potential scope and scope are line 6–7.

Breaking down the scopes into lines is in general not possible, of course, since line breaks may (or may not) appear almost anywhere. If we want to talk about scope on a line-by-line basis, we have to format the program accordingly.

**Storage duration.** Related to the scope of a variable is its *storage duration*. This term denotes the time in which the address of the variable is valid, that is, some memory location is assigned to it.

For a variable with local scope, the storage duration is usually the time in which the program’s control is in the variable’s potential scope. During program execution, this means that whenever the variable declaration is reached, some memory location is assigned and the address becomes valid. And whenever the execution gets to the end of the declarative region, the associated memory is freed and the variable’s address becomes
2.4. CONTROL STATEMENTS

Figure 5: Potential scopes of declarations $D, E_1, E_2, E_3$ of the same name, drawn as rectangles with the corresponding declaration in the upper left corner (left); on the right, we see the resulting scopes of $D$ (dark gray), $E_1, E_3$ (light gray) and $E_2$ (white).

invalid. We therefore get a “fresh instance” of the variable everytime its declaration is executed.

This behavior is called automatic storage duration. For example, in

```cpp
for (unsigned int i = 0; i < 10; ++i) {
    const int k = i;
    ...
}
```

the initialization takes place in each iteration, and it yields a different value each time. This is fine, since the “constness” individually refers to each of the ten instances of $k$ that are being generated throughout the loop.

As a more concrete example, consider the following code fragment.

```cpp
1   int i = 5;
2   for (int j = 0; j < 5; ++j) {
3       std::cout << ++i; // outputs 6, 7, 8, 9, 10
4       int k = 2;
5       std::cout << --k; // outputs 1, 1, 1, 1, 1
6   }
```

Since line 3 belongs to the scope of the declaration in line 1, the effect of line 3 is to increment the variable defined in line 1 in every iteration of the `for` statement. Line 5, on the other hand, belongs to the scope of the declaration in line 4; the effect of line 5 is therefore to decrement the “fresh” variable $k$ in every iteration, and this always results in value 1.

In contrast, a variable that is defined in namespace scope or global scope has static
storage duration. This means that its address is determined at the beginning of the program’s execution, and it does not change (hence “static”) nor become invalid until the execution of the program ends. The variables named by std::cin and std::cout, for instance, have static storage duration. Variables with static storage duration are also referred to as static variables.

2.4.4 Iteration: while statements

So far, we have seen one iteration statement, the for statement. The while statement is a simplified for statement, where both init-statement and expression are omitted. Its syntax is

```cpp
while ( condition )
  statement
```

where condition and statement are as in a for statement. As before, statement is referred to as the body of the while statement. Semantically, a while statement is equivalent to the corresponding for statement

```cpp
for ( ; condition ; )
  statement
```

The execution order is therefore condition, statement, condition, . . . until condition returns false.

Since while statements are so easy to rewrite as for statements, why do we need them? The main reason is readability. As its name suggests, a for statement is typically perceived as a counting loop in which the increment (or decrement) of a single variable is responsible for the progress towards termination. In this case, the progress is most conveniently made in the for statement’s expression. But the situation can be more complex: the progress may depend on the values of several variables, or on some condition that we check in the loop’s body. In some of these cases, a while statement is preferable. The next section describes an example.

The Collatz problem. Given a natural number \( n \in \mathbb{N} \), we consider the Collatz sequence \( n_0, n_1, n_2, \ldots \) with \( n_0 = n \) and

\[
n_i = \begin{cases} 
n_{i-1}/2, & \text{if } n_{i-1} \text{ is even} \\
3n_{i-1} + 1, & \text{if } n_{i-1} \text{ is odd}
\end{cases} \quad i \geq 1.
\]

For example, if \( n = 5 \), we get the sequence 5, 16, 8, 4, 2, 1, 4, 2, 1, . . . . Since the sequence gets repetitive as soon as 1 appears, we may stop at this point. Program 10 reads in a number \( n \) and outputs the elements of the sequence \( (n_i)_{i \geq 1} \) until the number 1 appears.
1 // Program: collatz.cpp
2 // Compute the Collatz sequence of a number n.
3 #include <iostream>
4 int main()
5 {
6   // Input
7     std::cout << "Compute the Collatz sequence for n =? ";
8     unsigned int n;
9     std::cin >> n;
10    // Iteration
11    while (n > 1) {
12       if (n % 2 == 0)
13         n = n / 2;
14       else
15         n = 3 * n + 1;
16       std::cout << n << " ";
17    }
18    std::cout << "\n"
19    return 0;
20 }

Program 10: ../progs/lecture/collatz.cpp

The loop can of course be written as a for statement with empty init-statement
and expression, but the resulting variant of the program is less readable since it tries to
advertise the rather complicated iteration as a simple counting loop. As a rule of thumb,
if there is a simple expression that captures the loop's progress, use a for statement.
Otherwise, consider formulating your loop as a while statement.

Talking about progress: is it clear that the number 1 always appears? If not, the
program collatz.cpp contains an infinite loop for certain values of n. If you play with
the program, you will observe that 1 indeed appears for all numbers you try, although
this may take a while. You will find, for example, that the Collatz sequence for n = 27 is

27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, 121,
364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175, 526,
263, 790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251, 754,
377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079,
3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154,
3077, 9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488,
244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4,
2, 1.
2.4.5 Iteration: do statements

Do statements are similar to while statements, except that the condition is evaluated after every iteration of the loop instead of before every iteration. Therefore, in contrast to for- and while statements, the body of a do statement is executed at least once.

The syntax of a do statement is as follows.

\[
\text{do}
\begin{align*}
& \text{statement} \\
& \text{while} (\text{expression});
\end{align*}
\]

where expression is of a type whose values can be converted to bool.

The semantics is defined as follows. An iteration of the loop consists of first executing statement and then evaluating expression. If expression returns true then another iteration follows. Otherwise, the do statement terminates. The execution order is therefore statement, expression, statement, expression, ... until expression returns false.

Alternatively, the semantics could be defined in terms of the following equivalent for statement.

\[
\text{for} (\text{bool firsttime = true; firsttime || expression; firsttime = false})
\begin{align*}
& \text{statement}
\end{align*}
\]

This behaves like our "simulation" of the while statement, except that in the first iteration, expression is not evaluated (due to short circuit evaluation, see Section 2.3.3), and statement is executed unconditionally.

Consider a simple calculator-type application in which the user enters a sequence of numbers, and after each number the program outputs the sum of the numbers entered so far. By entering 0, the user indicates that the program should stop. This is most naturally written using a do statement, since the termination condition can only be checked after the next number has been entered.

```cpp
int a; // next input value
int s = 0; // sum of values so far
do {
    std::cout << "next number =? ";
    std::cin >> a;
    s += a;
    std::cout << "sum = " << s << "\n";
} while (a != 0);
```
In this case, it is not possible to declare a where we would usually do it, namely immediately before the input statement. The reason is that a would then be local to the body of the do statement and would not be visible in the do statement’s expression a != 0.

2.4.6 Jump statements

At this point, we would like to extend our arsenal of control statements with a special type of statements that are referred to as jump statements. These statements are not necessary in the sense that they would allow you to do something which is not possible otherwise. Instead, just like while- and do statements (which are also unnecessary in that sense), jump statements provide additional flexibility in designing iteration statements. You should use this flexibility wherever it allows you to improve your code. However, be also warned that jump statements should be used with care since they tend to complicate the control flow. The complication of the control flow has to be balanced by a significant gain in other categories (such as code readability). Therefore, think carefully before introducing a jump statement!

When a jump statement is executed, the program flow unconditionally “jumps” to a certain point. There are two different jump statements that we want to discuss here.

The first jump statement is called a break statement; its syntax is rather simple.

```
break;
```

When a break statement is executed within an iteration statement, the smallest enclosing iteration statement terminates immediately. The execution continues at the statement after the iteration statement (if any). For example,

```
for (;;) break;
```

is not an infinite loop but rather a complicated way of writing a null statement. Here is a more useful appearance of break. In our calculator example from Page 92 it would be more elegant to suppress the irrelevant addition of 0 in the last iteration. This can be done with the following loop.

```
for (;;) {
    std::cout << "next number =? ";
    std::cin >> a;
    if (a == 0) break;
    s += a;
    std::cout << "sum = " << s << "\n";
}
```

Here, we see the typical usage of break, namely the termination of a loop “somewhere in the middle”. Note that we could equivalently write

```
do {
std::cout << "next number =? ";
```
std::cin >> a;
if (a == 0) break;
s += a;
std::cout << "sum = " << s << "\n";
} while (true);

In this case for is preferable, though, since it nicely reads as “forever”. Of course, the same functionality is possible without break, but the resulting code requires an additional block and evaluates a != 0 twice.

do {
    std::cout << "next number =? ";
    std::cin >> a;
    if (a != 0) {
        s += a;
        std::cout << "sum = " << s << "\n";
    }
} while (a != 0);

The second jump statement is called a continue statement; again the syntax is simple.

continue;

When a continue statement is executed, the remainder of the smallest enclosing iteration statement’s body is skipped, and execution continues at the end of the body. The iteration statement itself is not terminated.

If the surrounding iteration statement is a while or do statement, the execution therefore continues by evaluating its condition (expression, respectively). If the surrounding iteration statement is a for statement, the execution continues by evaluating its expression and then its condition. Like the break statement, the continue statement can therefore be used to manipulate the control flow “in the middle” of a loop.

In our calculator example, the following variant of the loop ignores negative input. Again, it would be possible to do this without continue, at the expense of another nested block.

for (;;) {
    std::cout << "next number =? ";
    std::cin >> a;
    if (a < 0) continue;
    if (a == 0) break;
s += a;
    std::cout << "sum = " << s << "\n";
}
2.4.7 Equivalence of iteration statements

In terms of pure functionality, the while- and do statements are redundant, as both of them can equivalently be expressed using a for statement. This may create the impression that for statements have more expressive power than while- and do statements. In this section we show that this is not the case: all three iteration statements are functionally equivalent. More precisely, we show how to use

- do statements to express while statements, and
- while statements to express for statements.

If we denote "A can be used to express B" by $A \Rightarrow B$, we therefore have

$$\text{do statement} \Rightarrow \text{while statement} \Rightarrow \text{for statement} \Rightarrow \text{do statement},$$

where we know the last implication from the previous section. Together, this clearly "proves" the claimed equivalence.

Note that we put the word *proves* in quotes, as our reasoning cannot be considered a formal proof. In order to really prove a statement like this, we first of all would have to be more formal in defining the semantics of statements. Semantics of programming languages is a subject of its own, and the formal treatment of semantics is way beyond what we can do here. In other words: The following is as much of a "proof" as you will get here, but it is sufficient to understand the relations between the three iteration statements.

**do statement $\Rightarrow$ while statement.** Consider the while statement

```plaintext
while ( condition )
  statement
```

Your first idea how to simulate this using a do statement might look like this:

```plaintext
if ( condition )
  do
    statement
  while ( condition );
```

Indeed, this induces the execution order `condition, statement, condition,... until condition returns false` and the statement terminates. But there is a simple technical problem: if `condition` is a variable declaration, we can't use it as the *expression* in the do statement. Here is a reformulation that works. (We are not saying that this should be done in practice. On the contrary, this should *never be done in practice. This section is about conceptual equivalence, not about practical equivalence. *)

```plaintext
do
  if ( condition )
```
This induces exactly the while statement's execution order condition, statement, condition,... until condition returns false and the loop is terminated using break.

### while statement ⇒ for statement

Simulating the for statement

```c
for ( init-statement condition; expression )
  statement
```

by a while statement seems easy:

```c
{
  init-statement
  while ( condition ) {
    statement
    expression;
  }
}
```

Indeed, this will work, unless statement contains a continue. In the for statement, execution would then proceed with the evaluation of expression, but in the simulating while statement, expression is skipped, and condition comes next. This reformulation is therefore wrong. Here is a version that works:

```c
{
  init-statement
  while ( condition ) {
    bool b = false;
    while ( b = !b )
      statement
    if ( b ) break;
    expression;
  }
}
```

This looks somewhat more complicated, so let us explain what is going on.

We may suppose that the identifier b does not appear in the given for statement (otherwise we choose a different name). Note that the whole statement forms a separate block, as does a for statement. A potential declaration in init-statement as well as the scope of b is thus limited to this block.

Consider an execution of the outer while statement. First, condition is evaluated, and if it returns false the statement terminates. Otherwise, the variable b is set to true in the inner while statement's condition, meaning that statement is executed next. (Recall
that the assignment operator returns the new value of its left operand.) If statement does not contain a break, the inner loop evaluates its condition for the second time. In doing so, b is set to false, and the condition returns false. Therefore, the inner loop terminates. Since b is now false, expression is evaluated next, followed by condition. This induces the for statement's execution order condition, statement, expression, condition, . . . until condition returns false and the outer loop terminates.

In the case where statement contains a break, the inner loop terminates immediately, and b remains true. In this case, we also terminate the outer loop that represents our original for statement.

In retrospect, we should now check that jump statements cause no harm in our previous reformulation of the while statement in terms of the do statement. We leave this as an exercise.

2.4.8 Choosing the "right" iteration statements

We have seen that from a functional point of view, the for statement, the while statement and the do statement are equivalent. Moreover, the break and continue statements are redundant. Still, C++ offers all of these statements, and this gives you the freedom (but also the burden) of choosing the appropriate control statements for your particular program.

Writing programs is a dynamic process. Even though the program may do what you want at some point, the requirements change, and you will keep changing the program in the future. Even if there is currently no need to change the functionality of the program, you may want to replace a complicated iteration statement by an equivalent simpler formulation. The general theme here is refactoring: the process of rewriting a program to improve its readability or structure, while keeping its functionality unchanged.

Here is a simple guideline for writing "good" loops. Choose the loop that leads to the most readable and concise formulation. This means

- few statements,
- few lines of code,
- simple control flow, and
- simple expressions.

Almost never there is the one and only best formulation; however, there are always arguably bad choices which you should try to avoid. Usually, there are some tradeoffs, like fewer lines of code versus more complicated expressions, and there is also some amount of personal taste involved. You should experience and find out what suits you best.

Let us look at some examples to show what we mean. Suppose that you want to output the odd numbers between 0 and 100. Having just learned about the continue statement, you may write the following loop.
for (unsigned int i = 0; i < 100; ++i) {
    if (i % 2 == 0) continue;
    std::cout << i << "\n";
}

This is perfectly correct, but the following version is preferable since it has fewer statements and fewer lines of code.

for (unsigned int i = 0; i < 100; ++i)
    if (i % 2 != 0) std::cout << i << "\n";

This variant still contains nested control statements; but you can get rid of the if statement and obtain code with simpler control flow.

for (unsigned int i = 1; i < 100; i += 2)
    std::cout << i << "\n";

The same output can be produced with a while statement and equally simple control flow.

int i = -1;
while ((i += 2) < 100)
    std::cout << i << "\n";

But here, the condition is more complicated, since it combines assignment and comparison operators. Such expressions are comparatively difficult to understand due to the effect of the assignment operation. Also, the initialization of $i$ to $-1$ is counter-intuitive, given that we deal with natural numbers.

You can solve the latter problem and at the same time get simpler expressions by writing

unsigned int i = 1;
while (i < 100) {
    std::cout << i << "\n";
    i += 2;
}

The price to pay is that you get less concise code; there are now five lines instead of the two lines that the for statement needs. It seems that for the simple problem of writing out odd numbers, a for statement with expression $i += 2$ is the loop of choice.

2.4.9 Details

Nested if-else statements. Consider the statement

if(true) if (false); else std::cout << "Where do I belong?";

It is not a priori clear what its effect is: if the else branch belongs to the outer if, there will be no output (since the condition has value true), but if the else branch belongs to the inner if, we get the output Where do I belong?
The intuitive rule is that the else branch belongs to the if immediately preceding it, in our case to the inner if. Therefore, the output is Where do I belong?, and we should actually format the statement like this:

```cpp
if(true)
    if (false)
        ; // null statement
else
    std::cout << "Where do I belong?";
```
Whenever you are unsure about rules like this, you can make the structure clear through explicit blocks:

```cpp
if(true) {
    if (false) {
        ; // null statement
    }
    else {
        std::cout << "Where do I belong?";
    }
}
```

The switch statement. Besides if...else there exists a second selection statement in C++: the switch statement. It is useful to select between many alternative statements, using the following syntax.

```cpp
switch ( condition )
{
    case literal:
        // statements
    default:
        // default statements
}
```

The value of condition must be convertible to an integral type. This is in contrast to the other control statements where condition has to be convertible to bool.

statement is usually a block that contains several labels of the form case literal:

where literal is a literal of integral type. For no two labels shall these literals have the same value. There can also be a label default.

The semantics of a switch statement is the following. condition is evaluated and the result is compared to each of the literals which appear in a label in statement. If for any of them the values agree, the execution continues at the statement immediately following the label. If there is no agreement but a default: label, the execution continues at the statement immediately following the default: label. Otherwise, statement is ignored and the execution continues after the switch statement.

Note that switch only selects an entry point for the processing of statement, it does not exit when the execution reaches another label. If one wants to separate the different alternatives, one has to use break (and this is the only legal use of break outside of
an iteration statement). Consider for example the following piece of code, and let us suppose that \( x \) is a variable of type `int`.

```cpp
switch (x) {
    case 0: std::cout << "0";
    case 1: std::cout << "1"; break;
    default: std::cout << "whatever";
}
```

For \( x = 0 \) the output is `01`; for \( x = 1 \) the output is `1`; otherwise we get the output `whatever`.

The `switch` statement is powerful in the sense that it allows the different alternatives to share code. However, this power also makes `switch` statements hard to read and error prone. A frequent problem is that one forgets to put a `break` where there should be one. Therefore, we mention `switch` here for completeness only. Whenever there are only a few alternatives to be distinguished, play it safe and use `if...else` rather than `switch`.

**The Halting Problem, Decidability, and Computability.** The halting problem is one of the fundamental problems in the theory of computation. Informally speaking, the problem is to decide (using an algorithm) whether a given program halts (terminates) when executed on a given input (program state). The term "program" may refer to a C++- program, but also to a program in any other common programming language.

To attack the problem formally, the British mathematician Alan Turing (1912-1954) defined in a seminal paper a "minimal" programming language; a program in this language is known as a *Turing machine*.

Turing proved that the halting problem is undecidable for Turing machines, but the same arguments can also be used to prove the same statement for C++ programs.

What does "undecidable" mean? We have seen a simple loop for which it was painfully evident that it is an infinite loop, haven't we? Yes, indeed one can decide the halting problem for many concrete programs. Undecidable means that (in a particular model of computation) there cannot be an algorithm that decides the halting problem for *all possible* programs.

Despite their simplicity, Turing machines are a widely accepted model of computation; in fact, just like machine language, Turing machines can do everything that C++ programs can do, except that they usually need a huge number of very primitive operations for that.

At the same time as Turing, the American mathematician Alonzo Church (1903-1995) developed a computational model called \( \lambda \)-calculus. As it turned out, his model is equivalent to Turing machines in terms of computational power. The Church-Turing thesis states that "every function that is naturally regarded as computable can be computed by a Turing machine". As there is no rigorous definition of what is "naturally regarded as computable", this statement is not a theorem but a hypothesis that cannot be proven mathematically. As of today, the hypothesis has not been disproved. In theoretical computer science the term *computable* used without further qualification is a synonym for "computable by a Turing machine" (equivalently, a C++ program).
Point of declaration. Our approach of defining potential scope and scope line by line is a simplification, even if the code is suitably formatted and we only have one declaration per line. The truth is that the point of declaration of \( i \) in

```plaintext
int i = 5;
```

is in the *middle* of the declaration, after the name \( i \) has appeared. The potential scope therefore does not include the full line, but only the part starting from `=`. This explains what happens in the following code fragment, but fortunately this is consistent with our line-by-line approach. In

```plaintext
1  int i = 5;
2  {
3   int i = i;
4  }
```

the name \( i \) after the `=` in line 3 refers to the declaration in line 3. Consequently, \( i \) is initialized with itself in this line, meaning that its value will be undefined, and not 5.

In other situations it may happen, though, that the appearance of a name in the declaration of the same name refers to a *previous* declaration of this name. For now, we can easily avoid such subtleties by the following rule: any declaration should contain the name to be declared once only.

The Collatz problem and the `?`-operator. The Collatz sequence goes back to the German mathematician Lothar Collatz (1910–1990) who studied it in the 1930’s. Several prizes have been offered to anyone who proves or disproves the conjecture that the number 1 appears in the Collatz sequence of every number \( n \geq 1 \). The famous Hungarian mathematician Paul Erdős offered $500, which is much by his standards (he used to offer much lower amounts for very difficult problems). Erdős said that “Mathematics is not yet ready for such problems”. Indeed, the conjecture is still unsolved.

We have presented the computation of the Collatz sequence as an application of the `while` statement, pointing out that the conditional change of \( n \) is too complicated to put it into a `for` statement’s expression. Well, that’s not exactly true: the designers of C, the precursor to C++, had a weakness for very compact code and came up with the `conditional operator` that allows us to simulate `if` statements by expressions. The syntax of this *ternary* operator (arity 3) is

```plaintext
condition ? expression1 : expression2
```

Here, `condition` is an expression of a type whose values can be converted to `bool`, and `expression1` and `expression2` are expressions. The semantics is as follows. First, `condition` is evaluated. If it returns `true`, `expression1` is evaluated, and its value is returned as the value of the composite expression. Otherwise (if `condition` returns `false`), `expression2` is evaluated, and its value is returned. The token `?` is a sequence point (see Section [2.2.10]), meaning that all effects of `condition` are processed before either `expression1` or `expression2` are evaluated.
Using the conditional operator, the loop of Program 10 could quite compactly be written as follows.

```c++
for ( ; n > 1; std::cout << (n % 2 == 0 ? n=n/2 : n=3*n+1) << " ");
```

We leave it up to you to decide whether you like this variant better.

Static variables. The discussion about storage duration above does not tell the whole story: it is also possible to define variables with local scope that have static storage duration.

This is done by prepending the keyword `static` to the variable declaration. For example, in

```c++
for ( int i = 0; i < 5; ++i) {
    static int k = i;
    k += i;
    std::cout << k << "\n";
}
```

the address of `k` remains the same during all iterations, and `k` is initialized to `i` once only, in the first iteration. The above piece of code will therefore output the sequence of values 0, 1, 3, 6, 10 (remember Gauss). Without the `static` keyword, the result would simply be the sequence of even numbers 0, 2, 4, 6, 8.

Static variables have been quite useful in C, for example to count how often a specific piece of code is executed; in C++, they are less important.

For variables of fundamental type the initial value may be undefined, as in the definition `int x;`. However, the value is undefined only if `x` has automatic storage duration. In contrast, variables with static storage duration are always zero-initialized, that is, filled with a "zero" of the appropriate type.

Jump statements. There are two more jump statements in C++ that we haven't discussed in this section. One of them is the `return` statement that you already know (Section 2.1.15): it may occur only in a function, and its execution lets the program flow jump to the end of the corresponding function body. The other jump statement is the `goto` statement, but since this one is rarely needed (and somewhat difficult to use), we omit it.

2.4.10 Goals

Dispositional. At this point, you should ...

1) know the syntax and semantics of `if...else`, `for`, `while`, and `do` statements;

2) understand the concepts block, selection, iteration, declarative region, scope, and storage duration;
3) understand the concept of an infinite loop and be aware of the difficulty of detecting such loops;
4) understand the conceptual equivalence of for-, while-, and do statements;
5) know the syntax and semantics of continue- and break statements;
6) know at least four criteria to judge the code quality of iteration statements.

Operational. In particular, you should be able to ...

(G1) check a given simple program (as defined below) for syntactical correctness and point out possible errors;
(G2) read and understand a given simple program and explain what happens during its execution;
(G3) find (potential) infinite loops in a given simple program;
(G4) find the matching declaration for a given identifier;
(G5) determine declarative region and scope of a given declaration;
(G6) reformulate a given for-, while-, or do statement equivalently using any of the other two statements;
(G7) compare the code quality of two given iteration statements and pick the one that is preferable (if any);
(G8) design simple programs for given tasks.

The term simple program refers to a program that consists of a main function in which up to four (possibly nested) iteration statements appear, plus some selection statements. Naturally, only the fundamental types and operations discussed in the preceding sections are used.

2.4.11 Exercises

Exercise 37 Correct all syntax errors in the program below. What does the resulting program output for the following inputs?
(a) -4 (b) 0 (c) 1 (d) 3

```cpp
#include <iostream>

int main() {
    unsigned int x = +1;
    { std::cin >> x; }
    for (int y = 0u; y < x) {
        std::cout << ++y;
    }
    return 0;
}
```
Exercise 38 What is the problem with the code below? Fix it and explain what the resulting code computes.

```c
1 unsigned int s = 0;
2 do {
3     int i = 1;
4     if (i % 2 == 1) s *= i;
5 } while (++i < 10);
```

Exercise 39 For each variable declaration in the following program give its declarative region and its scope in the form “line x–y”. What is the output of the program?

```c
1 #include <iostream>
2 int main()
3 {
4     int s = 0;
5     {
6         int i = 0;
7         while (i < 4)
8             {
9             ++i;
10             const int f = i + 1;
11             s += f;
12             const int s = 3;
13             i += s;
14         }
15         const unsigned int t = 2;
16         std::cout << s + t << "\n";
17     }
18     const int k = 1;
19     return 0;
20 }
```

Exercise 40 Consider the program given below for each of the listed input numbers. Determine the values of x, s, and i at begin of the first five iterations of the for-loop, before the condition is evaluated. What does the program output for these inputs? (a) -1 (b) 1 (c) 2 (d) 3

```c
1 #include <iostream>
2 int main()
3 {
4     int x;
5     std::cin >> x;
6     int s = 0;
7     for (int i = 0; i < x; ++i) {
```
# include <iostream >
int main ()
{
    unsigned int x;
    std::cin >> x;
    unsigned int y = x;
    for (unsigned int s = 0; y >= 0; --y)
        s += y;
    std::cout << "s=" << s << "\n";
    return 0;
}

Exercise 41 Find at least four problems in the code given below. (G3)(G4)(G5)

Exercise 42 For which input numbers is the output of the program given below well defined? List those input/output pairs and argue why your list is complete. (G3)(G4)(G5)

Exercise 43 Reformulate the code below equivalently in order to improve its readability. Describe the program’s output as a function of its input n. (G2)(G6)(G7)
Exercise 44 Reformulate the program below equivalently in order to improve its readability and efficiency. Describe the program’s output as a function of its input $x$. 

```
#include <iostream>

int main()
{
  int x;
  std::cin >> x;
  int s = 0;
  int i = -10;
  do
    for (int j = 1;;)
      if (j++ < i) s += j - 1; else break;
  while (++i <= x);
  std::cout << s << "\n";
  return 0;
}
```

Exercise 45 What is the behavior of Program 8 on Page 85 if the user inputs 0 or 1? Rewrite the program (if this is necessary at all) so that it correctly handles all possible inputs (we adopt the convention that 0 and 1 are not prime numbers).

Exercise 46 Write a program fak-1.cpp to compute the factorial $n!$ of a given input number $n$.

Exercise 47 Write a program dec2bin.cpp that inputs a natural number $n$ and outputs the binary digits of $n$ in reverse order. For example, for $n=2$ the output is 01 and for $n=11$ the output is 1101 (see also Exercise 57).

Exercise 48 Write a program cross_sum.cpp that inputs a natural number $n$ and outputs the sum of the (decimal) digits of $n$. For example, for $n=10$ the output is 1 and for $n=112$ the output is 4.

Exercise 49 Write a program perfect.cpp to test whether a given natural number $n$ is perfect. A number $n \in \mathbb{N}$ is called perfect if and only if it is equal to the sum of its proper divisors, that is, $n = \sum_{k \in \mathbb{N}, s.t. k < n} k$. For example, $28 = 1 + 2 + 4 + 7 + 14$ is perfect, while $12 < 1 + 2 + 3 + 4 + 6$ is not.
Extend the program to find all perfect numbers between 1 and \( n \). How many perfect numbers exist in the range \([1, 50000]\)?

**Exercise 50** Write a program `dec2bin2.cpp` that inputs a natural number \( n \) and outputs the binary digits of \( n \) in the correct order. For example, for \( n=2 \) the output is 10 and for \( n=11 \) the output is 1011 (see also Exercise 47).

**Exercise 51** Pete and Colin play a dice game against each other. Pete has three four-sided (pyramidal) dice, each with faces numbered 1, 2, 3, 4. Colin has two six-sided (cubical) dice, each with faces numbered 1, 2, 3, 4, 5, 6. Peter and Colin roll their dice and compare totals: the highest total wins. The result is a draw if the totals are equal.

What is the probability that Pyramidal Pete beats Cubic Colin? What is the probability that Cubic Colin beats Pyramidal Pete? And what is the probability of a draw? As a consequence, is it a fair game, and if not, who would you rather be?

Write a program `dice.cpp` that outputs the aforementioned probabilities as rational numbers of the form \( p/q \). (This is a simplified version of Problem 205 from the Project Euler, see http://projecteuler.net/).

**Exercise 52** We know from Section 1.1 that it took Frank Nelson Cole around three years to find the factorization

\[
761838257287 = 193707721
\]

of the Mersenne number \( 2^{67} - 1 \) by hand calculations. Write a program `cole.cpp` that performs the same task (hopefully in less than three years).

**Hint:** You will need the type `ifm::integer`, see Section 2.1.16.

**2.4.12 Challenges**

**Exercise 53** The Ulam spiral is named after the Polish mathematician Stanislaw Ulam who discovered it accidentally in 1963 while being bored during a scientific meeting.

The Ulam spiral is a method of “drawing” prime numbers. For this, one first lists all natural numbers in a spiral fashion, so that the integer grid \( \mathbb{Z}^2 \) gets filled with the natural numbers (see Figure 7(a)). For the drawing, every grid point containing a prime number is replaced with a black pixel, while the composite numbers are replaced with white pixels (Figure 7(b)). When this is done for the first \( n \) numbers (\( n \) large), one observes a surprising phenomenon: the prime numbers tend to accumulate along diagonals (see Figure 7(c)).

Write a program that outputs the Ulam spiral generated by the first \( n \) numbers, where \( n \) is read from the input. You may restrict \( n \) such that the drawing fits into a window of 500 \( \times \) 500 pixels, say.

**Hint:** You may use the `libwindow` library to produce the drawing. The example program in its documentation should give you an idea how this can be done.
Exercise 54 The n-queens problem is to place n queens on an n x n chessboard such that no two queens threaten each other. Formally, this means that there is no horizontal, vertical, or diagonal with more than one queen in it. Write a program that outputs the number of different solutions to the n-queens problem for a given input n. Assuming a 32 bit system, the program should work up to n = 9 at least. Check through a web search whether the numbers that your program computes are correct.

Exercise 55 The largest Mersenne prime known as of September 2009 is

$$2^{43,112,609} - 1,$$

see Section 1.1. In Exercise 19 we have asked you to find the number of decimal digits that this number has. In this challenge, we originally wanted to ask you to list all these digits, but in the interest of the TA that has to mark your solution, we decided to switch to the following variant: Write a program famous_last_digits.cpp that computes and outputs the last 10 decimal digits of the above Mersenne prime!
2.5 Floating point numbers

Furthermore, it has revealed the ratio of the chord and arc of ninety degrees, which is as seven to eight, and also the ratio of the diagonal and one side of a square which is as ten to seven, disclosing the fourth important fact, that the ratio of the diameter and circumference is as five-fourths to four.

\[
\frac{2\sqrt{2}}{\pi} = \frac{7}{8}, \quad \sqrt{2} = \frac{10}{7}, \quad \text{and} \quad \frac{1}{\pi} = \frac{5}{16} (1897)
\]

This section discusses the floating point number types float and double for approximating real numbers. You will learn about floating point number systems in general, and about the IEEE standard 754 that describes two specific floating point number systems. We will point out the strengths and weaknesses of floating point numbers and give you three guidelines to avoid common pitfalls in computing with floating point numbers.

When converting degrees Celsius into Fahrenheit with the program fahrenheit.cpp in Section 2.2 we make mistakes. For example, 28 degrees Celsius are 82.4 degrees Fahrenheit, but not 82 as output by fahrenheit.cpp. The reason for this mistake is that the integer division employed in the program simply “cuts off” the fractional part. What we need is a type that allows us to represent and compute with fractional numbers like 82.4.

For this, C++ provides two floating point number types float and double. Indeed, if we simply replace the declaration int celsius in fahrenheit.cpp by float celsius, the resulting program outputs 82.4 for an input value of 28. Floating point numbers also solve another problem that we had with the types int and unsigned int: float and double have a much larger value range and are therefore suitable for “serious” computations. In fact, computations with floating point numbers are very fast on modern platforms, due to specialized processors.

Fixed versus floating point. If you think about how to represent decimal numbers like 82.4 using a fixed number of decimal digits (10 digits, say), a natural solution is this: you partition the 10 available digits into 7 digits before the decimal point, say, and 3 digits after the decimal point. Then you can represent all decimal numbers of the form

\[
\sum_{i=-3}^{6} \beta_i 10^i,
\]

with \( \beta_i \in \{0, \ldots, 9\} \) for all \( i \). This is called a fixed point representation.
There are, however, two obvious disadvantages of a fixed point representation. On the one hand, the value range is very limited. We have already seen in Section 2.2.5 that the largest int value is so small that it hardly allows any interesting computations (as an example, try out Program 1 on some larger input). A fixed point representation is even worse in this respect, since it reserves some of our precious digits for the fractional part after the decimal point, even if these digits are not—or not fully—needed (as in 82.4).

The second disadvantage is closely related: even though the two numbers 82.4 and 0.0824 have the same number of significant digits (namely 3), the latter number is not representable with only 3 digits after the decimal point. Here, we are wasting the 7 digits before the decimal point, but we are lacking digits after the decimal point.

A floating point representation resolves both issues by representing a number simply as its sequence of decimal digits (an integer called the significand), plus the information “where the decimal point is”. Technically, one possibility to realize this is to store an exponent such that the represented number is of the form

\[ \text{significand} \cdot 10^{\text{exponent}}. \]

For example,

\[ 82.4 = 824 \cdot 10^{-1}, \]
\[ 0.0824 = 824 \cdot 10^{-4}. \]

2.5.1 The types float and double

The types float and double are fundamental types provided by C++, and they store numbers in floating point representation.

While the fundamental types int and unsigned int are meant to approximate the “mathematical types” $\mathbb{Z}$ and $\mathbb{N}$, respectively, the goal of both float and double is to approximate the set $\mathbb{R}$ of real numbers. Since there are much more real numbers than integers, this goal seems even more ambitious (and less realistic) than trying to approximate $\mathbb{Z}$, say, with a finite value range. Nevertheless, the two types float and double are very useful in practical applications. The floating point representation allows values that are much larger than any value of type int and unsigned int. In fact, the value ranges of the floating point number types float and double are sufficient in most applications.

Values of these two types are referred to as floating point numbers, where double usually allows higher (namely, double) precision in approximating real numbers.

On the types float and double we have the same arithmetic, relational, and assignment operators as on integral types, with the same associativities and precedences. The only exception is that the modulus operators $\%$ and $\%=$ are available for integral types only. This makes sense, since division over float and double is meant to model the true division over $\mathbb{R}$ which has no remainder.
Like integral types, the floating point number types are arithmetic types, and this completes the list of fundamental arithmetic types in C++.

Literals of type float and double. Literals of types float and double are more complicated than literals of type int or unsigned int. For example, 1.23e-7 is a valid double literal, representing the value $1.23 \cdot 10^{-7}$. Literals of type float look the same as literals of type double, followed by the letter f or F.

In its most general form, a double literal consists of an integer part, followed by a fractional part (starting with the decimal point .), and an exponential part (starting with the letter e or E). The literal 1.23e-7 has all of these parts.

Both the integer part as well as the fractional part (after the decimal point) are sequences of digits from 0 to 9, where one of them may be empty, like in .1 (meaning 0.1) and in 1. (meaning 1.0). The exponential part (after the letter e or E) is also a sequence of digits, preceded by an optional + or -. Either the fractional part or the exponential part may be omitted. Thus, 123e-9 and 1.23 are valid double literals, but 123 is not, in order to avoid confusion with int literals.

The value of the literal is obtained by scaling the fractional decimal value defined by the integer part and the fractional part by $10^e$, where e is the (signed) decimal integer in the exponential part (defined as 0, if the exponential part is missing).

To show floating point numbers in action, let us write a program that "computes" a fully-fledged real number, namely the Euler constant

$$\sum_{i=0}^{\infty} \frac{1}{i!} = 2.71828\ldots$$

You may recall that this sum converges quickly, so we should already get a good approximation for the Euler constant when we sum up the first 10 terms, say. Program 11 does exactly this.

```cpp
// Program: euler.cpp
// Approximate Euler's constant e.

#include <iostream>

int main ()
{
    // values for term i, initialized for i = 0
    float t = 1.0f; // 1/i!
    float e = 1.0f; // i-th approximation of e

    std::cout << "Approximating the Euler constant...\n";
    // steps 1,...,n
    for (unsigned int i = 1; i < 10; ++i) {
```
When you run the program, its output may look like this.

Approximating the Euler constant...
Value after term 1: 2
Value after term 2: 2.5
Value after term 3: 2.66667
Value after term 4: 2.70833
Value after term 5: 2.71667
Value after term 6: 2.71806
Value after term 7: 2.71825
Value after term 8: 2.71828
Value after term 9: 2.71828

It seems that we do get a good approximation of the Euler constant in this way. What remains to be explained is how the mixed expression \( e += t /= i \) in line 15 is dealt with that contains operands of types unsigned int and float. Note that since the arithmetic assignment operators are right-associative (Table 1 on Page 51), this expression is implicitly parenthesized as \( e += (t /= i) \). When evaluated in iteration \( i \), it therefore first divides \( t \) by \( i \) (corresponding to the step from \( 1/(i-1)! \) to \( 1/i! \)), and then it adds the resulting value \( 1/i! \) to the approximation \( e \).

2.5.2 Mixed expressions, conversions, and promotions

The floating point number types are defined to be more general than any integral type. Thus, in mixed composite expressions, integral operands get converted to the respective floating point number type (see also Section 2.2.7 where we first saw this mechanism for mixed expressions over the types int and unsigned int). The resulting value is the representable value nearest to the original value, where ties are broken in an implementation-defined fashion. In particular, if the original integer value is in the value range of the relevant floating point number type, the value remains unchanged.

This in particular explains why the change of int celsius to float celsius in the program fahrenheit.cpp leads to the behavior we want: during evaluation of the expression \( 9 *\) celsius / 5 + 32, all integral operands are eventually converted to float, so that the computation takes place exclusively over the type float.

In the program euler.cpp, we have the same kind of conversion: in the mixed expression \( t /= i \), the unsigned int operand \( i \) gets converted to the type float of the other operand \( t \).
2.5. FLOATING POINT NUMBERS

The type double is defined to be more general than the type float. Thus, a composite expression involving operands of types float and double is of type double. When such an expression gets evaluated, every operand of type float is promoted to double. Recall from Section 2.3.2 that promotion is a term used to denote certain privileged conversions in which no information gets lost. In particular, the value range of double must contain the value range of float.

In summary, the hierarchy of arithmetic types from the least general to the most general type is

\[
\text{bool} \prec \text{int} \prec \text{unsigned int} \prec \text{float} \prec \text{double}.
\]

We already know that a conversion may also go from the more general to the less general type, see Section 2.2.7. This happens for example in the declaration statement

\[
\text{int } i = -1.6f;
\]

When a floating point number is converted to an integer, the fractional part is discarded. If the resulting value is in the value range of the target type, we get this value, otherwise the result of the conversion is undefined. In the previous example, this rule initializes \(i\) with \(-1\) (and not with the nearest representable value \(-2\)).

When double values are converted to float, we again get the nearest representable value (with ties broken in an implementation-dependent way), unless the original value is larger or smaller than any float value. In this latter case, the conversion is undefined.

2.5.3 Explicit conversions

Conversions between integral and floating point number types are common in practice. For example, the conversion of a nonnegative float value \(x\) to the type unsigned int corresponds to the well-known floor function \([x]\) that rounds down to the next integer. Conversely, it can make sense to perform an integral computation over a floating point number type, if this latter type has a larger value range.

Explicit conversion allows to convert a value of any arithmetic type directly into any other arithmetic type, without the detour of defining an extra variable like in

\[
\text{int } i = -1.6f; \quad \text{To obtain the int value resulting from the float value } -1.6, \text{ we can simply write the expression } \text{int}(-1.6f).
\]

The general syntax of an explicit conversion, also called a cast expression, is

\[
T( \text{expr} )
\]

where \(T\) is a type, and \(\text{expr}\) is an expression. The cast expression is valid if and only if the corresponding conversion of \(\text{expr}\) to the type \(T\) (as in \(T \times = \text{expr}\)) is defined.

For certain “complicated” type names \(T\), it is necessary to parenthesize \(T\), like in the cast expression \((\text{unsigned int})(1.6f)\).
2.5.4 Value range

For integral types, the arithmetic operations may fail to compute correct results only due to over- or underflow. This is because the value range of each integral type is a contiguous subset of \( \mathbb{Z} \), with no "holes" in between.

For floating point number types, this is not true: with finite (and even with countable) value range, it is impossible to represent a subset of \( \mathbb{R} \) with more than one element but no holes. In contrast, over- or underflows are less of an issue: the representable values usually span a huge interval, much larger than for integral types. If you print the largest double value on your platform via the expression

\[
\text{std::numeric_limits<double>::max()}
\]

you might for example get the output \( 1.79769e+308 \). Recall that this means \( 1.79769 \times 10^{308} \), a pretty large number.

Let us approach the issue of holes with a very simple program that asks the user to input two floating point numbers and their difference. The program then checks whether this is indeed the correct difference. Program 12 performs this task.

---

Program 12: `../progs/lecture/diff.cpp`

```
1 // Program: diff.cpp
2 // Check subtraction of two floating point numbers
3
4 #include <iostream>
5
6 int main()
7 {
8     // Input
9     float n1;
10     std::cout << "First number =? ";
11     std::cin >> n1;
12
13     float n2;
14     std::cout << "Second number =? ";
15     std::cin >> n2;
16
17     float d;
18     std::cout << "Their difference =? ";
19     std::cin >> d;
20
21     // Computation and output
22     std::cout << "Computed difference - input difference = " << n1 - n2 - d << ".\n";
23     return 0;
24 }
```
Here is an example run showing that the authors are able to correctly subtract 1 from 1.5.

First number = 1.5
Second number = 1.0
Their difference = 0.5
Computed difference - input difference = 0.

But the authors can apparently not correctly subtract 1 from 1.1:

First number = 1.1
Second number = 1.0
Their difference = 0.1
Computed difference - input difference = 2.23517 e-08.

What is going on here? After double checking our mental arithmetic, we must conclude that it’s the computer and not us who cannot correctly subtract. To understand why, we have to take a somewhat closer look at floating point numbers in general.

2.5.5 Floating point number systems

A finite floating point number system is a finite subset of \( \mathbb{R} \), defined by four numbers \( 2 \leq \beta \in \mathbb{N} \) (the base), \( 1 \leq p \in \mathbb{N} \) (the precision), \( e_{\text{min}} \in \mathbb{Z} \) (the smallest exponent) and \( e_{\text{max}} \in \mathbb{Z} \) (the largest exponent).

The set \( \mathcal{F}(\beta, p, e_{\text{min}}, e_{\text{max}}) \) of real numbers represented by this system consists of all floating point numbers of the form

\[
s \cdot \sum_{i=0}^{p-1} d_i \beta^{-i} \cdot \beta^e,
\]

where \( s \in \{-1, 1\} \), \( d_i \in \{0, \ldots, \beta - 1\} \) for all \( i \), and \( e \in \{e_{\text{min}}, \ldots, e_{\text{max}}\} \).

The number \( s \) is the sign, the sequence \( d_0 d_1 \ldots d_{p-1} \) is called the significand (an older equivalent term is mantissa), and the number \( e \) is the exponent.

We sometimes write a floating point number in the form

\[
\pm d_0 . d_1 \ldots d_{p-1} \cdot \beta^e.
\]

For example, using base \( \beta = 10 \), the number 0.1 can be written as \( 1.0 \cdot 10^{-1} \), and as \( 0.1 \cdot 10^0, 0.01 \cdot 10^1 \) and in many other ways.

The representation of a number becomes unique when we restrict ourselves to the set \( \mathcal{F}^*(\beta, p, e_{\text{min}}, e_{\text{max}}) \) of normalized numbers, i.e. the ones with \( d_0 \neq 0 \). The downside of this is that we lose some numbers (in particular the number 0, but let’s not worry about this now). More precisely, normalization loses exactly the numbers of absolute value smaller than \( \beta^{e_{\text{min}}} \) (see also Exercise 62).

For a fixed exponent \( e \), the smallest positive normalized number is

\[
1.0 \ldots 0 \cdot \beta^e = \beta^e,
\]
while the largest one is (by the formula \( \sum_{i=0}^{n} x^i = (x^{n+1} - 1)/(x - 1) \) for \( x \neq 1 \))

\[
(\beta - 1)(\beta - 1) \ldots (\beta - 1) \cdot \beta^e = \sum_{i=0}^{p-1} (\beta - 1)\beta^{-i} \cdot \beta^e = \left(1 - \left(\frac{1}{\beta}\right)^p\right) \beta^{e+1} < \beta^{e+1}.
\]

This means that the normalized numbers are “sorted by exponent”.

Most floating point number systems used in practice are binary, meaning that they have base \( \beta = 2 \). In a binary system, the decimal numbers 1.1 and 0.1 are not representable, as we will see next; consequently, errors are made in converting them to floating point numbers, and this explains the strange behavior of Program 12.

Computing the floating point representation. In order to convert a given positive decimal number \( x \) into a normalized binary floating point number system \( \mathcal{F}^*(2, p, e_{\min}, e_{\max}) \), we first compute its binary expansion

\[
x = \sum_{i=-\infty}^{\infty} b_i 2^i, \quad b_i \in \{0, 1\} \text{ for all } i.
\]

This is similar to the binary expansion of a natural number as discussed in Section 2.2.8. The only difference is that we have to allow all negative powers of 2, since \( x \) can be arbitrarily close to 0. The binary expansion of 1.25 for example is

\[
1.25 = 1 \cdot 2^{-2} + 1 \cdot 2^0.
\]

We then determine the smallest and largest values of \( \bar{i}, \underline{i} \) and \( \bar{e} \), for which \( b_i \) is nonzero (note that \( \underline{i} \) may be \( -\infty \), but \( \bar{e} \) is finite since \( x \) is finite). The number \( \bar{e} - \underline{i} + 1 \in \mathbb{N} \cup \{\infty\} \) is the number of significant digits of \( x \).

With \( d_i := b_{\bar{i}-i} \), we get \( d_0 \neq 0 \) and

\[
x = \sum_{i=\bar{i}}^{\infty} b_i 2^i = \sum_{i=0}^{\bar{e}-\underline{i}} b_{\bar{e}-i} 2^{\bar{e}-i} = \sum_{i=0}^{\bar{e}-\underline{i}} d_i 2^{-i} \cdot 2^{\bar{e}}.
\]

This implies that \( x \in \mathcal{F}^*(2, p, e_{\min}, e_{\max}) \) if and only if \( \bar{e} - \underline{i} < p \) and \( e_{\min} \leq \bar{e} \leq e_{\max} \). Equivalently, if the binary expansion of \( x \) has at most \( p \) significant digits, and the exponent of the normalized representation is within the allowable range.

In computing the binary expansion of \( x > 0 \), let us assume for simplicity that \( x < 2 \). This is sufficient to explain the issue with the decimal numbers 1.1 and 0.1, and all other cases can be reduced to this case by separately dealing with the largest even integer smaller or equal to \( x \): writing \( x = y + 2k \) with \( k \in \mathbb{N} \) and \( y < 2 \), we get the binary expansion of \( x \) by combining the expansions of \( y \) and \( 2k \).

For \( x < 2 \), we have

\[
x = \sum_{i=-\infty}^{0} b_i 2^i = b_0 + \sum_{i=0}^{-1} b_i 2^i = b_0 + \sum_{i=-\infty}^{0} b_{\bar{i}-1} 2^{\bar{i}-1} = b_0 + \frac{1}{2} \sum_{i=-\infty}^{0} b_{\bar{i}-1} 2^{\bar{i}}.
\]
This identity provides a simple algorithm to compute the binary expansion of $x$. If $x \geq 1$, the most significant digit $b_0$ is 1, otherwise it is 0. The other digits $b_i$, $i \leq -1$, can subsequently be extracted by applying the same technique to $x' = 2(x - b_0)$.

Doing this for $x = 1.1$ yields the following sequence of digits.

$$
\begin{align*}
1.1 & \rightarrow b_0 = 1 \\
2(1.1 - 1) &= 2 \cdot 0.1 = 0.2 \rightarrow b_{-1} = 0 \\
2(0.2 - 0) &= 2 \cdot 0.2 = 0.4 \rightarrow b_{-2} = 0 \\
2(0.4 - 0) &= 2 \cdot 0.4 = 0.8 \rightarrow b_{-3} = 0 \\
2(0.8 - 0) &= 2 \cdot 0.8 = 1.6 \rightarrow b_{-4} = 1 \\
2(1.6 - 1) &= 2 \cdot 0.6 = 1.2 \rightarrow b_{-5} = 1 \\
2(1.2 - 1) &= 2 \cdot 0.2 = 0.4 \rightarrow b_{-6} = 0 \\
\vdots
\end{align*}
$$

We now see that the binary expansion of the decimal number 1.1 is periodic: the corresponding binary number is 1.00011, and it has infinitely many significant digits. Since all numbers in the floating point number systems $\mathcal{F}(2, p, e_{\text{min}}, e_{\text{max}})$ and $\mathcal{F}^*(2, p, e_{\text{min}}, e_{\text{max}})$ have at most $p$ significant digits, it follows that $x = 1.1$ is not representable in a binary floating point number system, regardless of $p, e_{\text{min}}$ and $e_{\text{max}}$. The same is true for $x = 0.1$.

The Excel 2007 bug. We have shown in the previous paragraph that it is impossible to convert some common decimal numbers (like 1.1 or 0.1) into binary floating-point numbers, without making small errors. This has the embarrassing consequence that the types float and double are unable to represent the values of some of their own literals.

Despite this problem, a huge number of decimal-to-binary conversions take place on computers worldwide, the minute you read this. For example, whenever you enter a number into a spreadsheet, you do it in decimal format. But chances are high that internally, the number is converted to and represented in binary floating-point format. The small errors themselves are usually not the problem; but the resulting “weird” floating-point numbers extremely close to some “nice” decimal value may expose other problems in the program.

A recent such issue that has received a lot of attention is known as the the Excel 2007 bug. Users have reported that the multiplication of 77.1 with 850 in Microsoft Excel does not yield 65,535 (the mathematically correct result) but 100,000.

Microsoft reacted to this by admitting the bug, but at the same time pointing out that the computed value is correct, and that the error only happens when this value is displayed in the sheet. But how can it happen that the nice integer value 65,535 is incorrectly displayed? Well, it doesn’t happen: when you multiply 65,535 with 1, for example, the result is correctly displayed as 65,535.

The point is that the computed value is not 65,535, but some other number extremely close to it. The reason is that a small but unavoidable error is made in converting the
decimal value 77.1 into the floating-point number system internally used by Excel: like 1.1 and 0.1, the number 77.1 has no finite binary representation.

This error can of course not be "repaired" by the multiplication with 850, so Excel gets a value only very close to 65,535. This would be acceptable, but exactly for this value (and 11 others, according to Microsoft), the display functionality has a bug. Naturally, if only 12 "weird" numbers out of all floating-point numbers are affected by this bug, it is easy not to detect the bug during regular tests.

While Microsoft earned quite some ridicule for the Excel 2007 bug (for which it quickly offered a fix), it should in all fairness be admitted that such bugs could still be hidden in software of other vendors as well.

Relative error. If we are not able to represent a real number $x$ exactly as a binary floating point number in the system $F^*(2, p, e_{\min}, e_{\max})$, it is natural to approximate it by the floating point number nearest to $x$. What is the error we make in this approximation?

Suppose that $x$ is positive and has binary expansion

$$x = \sum_{i=-\infty}^{i} b_i 2^i = b_T . b_{T-1} \ldots 2^T,$$

where $b_T = 1$.

There are two natural ways of approximating $x$ with $p$ or less significant digits. One way is to round down, resulting in the number

$$\bar{x} = b_T . b_{T-1} \ldots b_{T-p+1} . 2^{T-p+1} = \sum_{i=T-p+1}^{T} b_i 2^i.$$

This truncates all the digits $b_i, i \leq T - p$, and the error we make is

$$x - \bar{x} = \sum_{i=-\infty}^{i=T-p} b_i 2^i \leq \sum_{i=-\infty}^{i=T-p} 2^i = 2^{T-p+1}.$$

Alternatively, we could round up to the number

$$\check{x} = \bar{x} + 2^{T-p+1}$$

where our previous error estimate shows that indeed, $x \leq \check{x}$ holds. For this, we have to check that $\check{x}$ has at most $p$ significant digits. This is true if $b_{T-p+1} = 0$, since then the addition of $2^{T-p+1}$ adds exactly one digit to the at most $p - 1$ significant digits of $\bar{x}$. And if $b_{T-p+1} = 1$, the addition of $2^{T-p+1}$ removes the least significant coefficient of $2^{T-p+1}$ and may create one extra carry digit at the other end.

This means that $x$ is between two numbers that are $2^{T-p+1}$ apart, so the nearer of the two numbers is at most $2^{T-p}$ away from $x$. On the other hand, $x$ has size at least $2^T$, meaning that

$$|x - \check{x}| / x \leq 2^{-p},$$
2.5. FLOATING POINT NUMBERS

where $\hat{x}$ is the floating point number nearest to $x$. The number $2^{-p}$, referred to as the *machine epsilon*, is the relative error made in approximating $x$ with its nearest floating point number $\hat{x}$.

The previous inequality also holds for negative $x$ and their corresponding best approximation $\hat{x}$, so that we get the general relative error formula

$$\frac{|x - \hat{x}|}{|x|} \leq 2^{-p}, \quad x \neq 0.$$  

This means that the distance of $x$ to its nearest floating point number is in the worst case proportional to the size of $x$. This is because the floating point numbers are not equally spaced along the real line. Close to 0, their density is high, but the more we go away from 0, the sparser they become. As a simple example, consider the normalized floating point number system $F^*(2, 3, -2, 2)$. The smallest positive number is $1.00 \cdot 2^{-2} = 1/4$, and the largest one is $1.11 \cdot 2^2 = 7$ (recall that the digits are binary). The distribution of all positive numbers over the interval $[1/4, 7]$ is shown in the following picture.

```
  0 1 2 3 4 5 6 7 8
```

From this picture, it is clear that the relative error formula cannot hold for very large $x$. But also if $x$ is very close to zero, the relative error formula may fail. In fact, there is a substantial gap between 0 and the smallest positive normalized number. Numbers $x$ in that gap are not necessarily approximable by normalized floating point numbers with relative error at most $2^{-p}$.

Where is the mistake in our calculations, then? There is no mistake, but the calculations are only applicable if the floating point number $\hat{x}$ nearest to $x$ is in fact a floating point number in the system we consider, i.e. if it has its exponent in the allowed range $\{e_{\text{min}}, \ldots, e_{\text{max}}\}$. This fails if $\hat{x}$ is too large or too small.

**Arithmetic operations.** Performing addition, subtraction, multiplication, and division with floating point numbers is easy in theory: as these are real numbers, we simply perform the arithmetic operations over the set $\mathbb{R}$ of real numbers; if the result is not representable in our floating point number system, we apply some rounding rule (such as choosing the nearest representable floating point number).

In practice, floating point number arithmetic is not more difficult than integer arithmetic. Let us illustrate this with an example. Suppose that $p = 4$, and that we have a binary system; we want to perform the addition

$$1.111 \cdot 2^{-2} + 1.011 \cdot 2^{-1}.$$
The first step is to align the two numbers such that they have the same exponent. This means to "denormalize" one of the two numbers, e.g. the second one:

\[ 1.111 \cdot 2^{-2} + 10.110 \cdot 2^{-2} \cdot 10.110 \cdot 2^{-2}. \]

Now we can simply add up the two significands, just like we add integers in binary representation. The result is

\[ 100.101 \cdot 2^{-2}. \]

Finally, we renormalize and obtain

\[ 1.00101 \cdot 2^0. \]

We now realize that this exact result is not representable with \( p = 4 \) significant digits, so we have to round. In this case, the nearest representable number is obtained by simply dropping the last two digits:

\[ 1.001 \cdot 2^0. \]

2.5.6 The IEEE standard 754

**Value range.** The C++ standard does not prescribe the value range of the types float and double. It only stipulates that the value range of float is contained in the value range of double such that a float value can be promoted to a double value.

In practice, most platforms support (variants of) the IEEE standard 754 for representing and computing with floating point numbers. Under this standard, the value range of the type float is the set

\[ F'(2, 24, -126, 127) \]

of *single precision* normalized floating point numbers, plus some special numbers (conveniently, 0 is one of these special numbers). The value range of double is the set

\[ F'(2, 53, -1022, 1023) \]

of *double precision* normalized floating point numbers, again with some special numbers added, including 0.

These parameters may seem somewhat arbitrary at first, but they are motivated by a common memory layout in which 32 bits can be manipulated at once. Indeed, 32 bits of memory are used to represent a single precision number. The significand requires 23 bits; recall that in a normalized binary floating point number system, the first digit of the significand is always 1, hence it need not explicitly be stored. The exponent requires another 8 bits for representing its \( 2^{10} - 1 \) possible values, and another bit is needed for the sign.
2.5. FLO\_ATING POINT NUMBERS

For double precision numbers, the significand requires 52 bits, the exponent has 11 bits for its $2046 = 2^{11} - 2$ possible values, and one bit is needed for the sign. In total, this gives 64 bits.

Note that in both cases, two more exponent values could be accommodated without increasing the total number of bits. These extra values are in fact used for representing the special numbers mentioned above, including 0.

Requirements for the arithmetic operations. The C++ standard does not prescribe the accuracy of arithmetic operations over the types float and double, but the IEEE standard 754 does. The requirements are as strict as possible: the result of any addition, subtraction, multiplication, or division is the representable value nearest to the true value. If there are two nearest values (meaning that the true value is halfway in between them), the one that has least significant digit $d_{p-1} = 0$ is chosen. This is called round-to-even; other rounding modes can be enabled if necessary. The same rule applies to the conversion of decimal values like 1.1 to their binary floating point representation.

Moreover, comparisons of values have to be exact under all relational operators (Section 2.3.2).

2.5.7 Computing with floating point numbers

We have seen that for every floating point number system, there are numbers that it cannot represent, and these are not necessarily very exotic, as our example with the decimal number 1.1 shows. On the other hand, the IEEE standard 754 guarantees that we will get the nearest representable number, and the same holds for the result of every arithmetic operation, up to (rare) over- and underflows. Given this, one might be tempted to believe that the results of most computations involving floating point numbers are close to the mathematically correct results, with respect to relative error.

Indeed, this is true in many cases. For example, our initial program euler.cpp computes a pretty good approximation of the Euler constant. Nevertheless, some care has to be taken in general. The goal of this section is to point out common pitfalls, and to provide resulting guidelines for “safe” computations with floating point numbers.

We start with the first and most important guideline that may already be obvious to you at this point.

| Floating Point Arithmetic Guideline 1: Never compare two floating point numbers for equality, if at least one of them results from inexact floating point computations. |

Even very simple expressions involving floating point numbers may be mathematically equivalent, but still return different values, since intermediate results are rounded. Two such expressions are $x * x - y * y$ and $(x + y) * (x - y)$. Therefore, testing the results of two floating point computations for equality using the relational operators $==$ or $!=$ makes little sense. Since equality is sensitive to the tiniest errors, we won’t get equality in most cases, even if mathematically, we would.
Given the formulation of the above guideline, you may wonder how to tell whether a particular floating point computation is exact or not. Exactness usually depends on the representation and is, therefore, hard to claim in general. However, there are certain operations which are easily seen to be exact. For instance, multiplication and division by a power of the base (usually, 2) do not change the significand, but only the exponent. Thus, these operations are exact, unless they lead to an over- or underflow in the exponent.

Moreover, it is safe to assume that the largest exponent is (much) higher than the precision \( p \), and in this case we can also exactly represent all integers of absolute value smaller than \( \beta^p \). Consequently, integer additions, subtractions, and multiplications within this range are exact.

The next two guidelines are somewhat less obvious, and we motivate them by first showing the underlying problem. Throughout, we assume a binary floating point number system of precision \( p \).

Adding numbers of different sizes. Suppose we want to add the two floating point numbers \( 2^p \) and 1. What will be the result? Mathematically, it is

\[
2^p + 1 = \sum_{i=0}^{p} b_i 2^i,
\]

with \((b_p, b_{p-1}, \ldots, b_0) = (1, 0, \ldots, 0, 1)\). Since this binary expansion has \( p + 1 \) significant digits, \( 2^p + 1 \) is not representable with precision \( p \). Under the IEEE standard 754, the result of the addition is \( 2^p \) (chosen from the two nearest candidates \( 2^p \) and \( 2^p + 2 \)), so this addition has no effect.

The general phenomenon here is that adding floating point numbers of different sizes “kills” the less significant digits of the smaller number (in our example, all its digits). The larger the size difference, the more drastic is the effect.

To convince you that this is not an artificial phenomenon, let us consider the problem of computing Harmonic numbers (search the web for the coupon collector’s problem to find an interesting occurrence of Harmonic numbers). For \( n \in \mathbb{N} \), the \( n \)-th Harmonic number \( H_n \) is defined as the sum of the reciprocals of the first \( n \) natural numbers, that is,

\[
H_n = \sum_{i=1}^{n} \frac{1}{i}.
\]

It should now be an easy exercise for you to write a program that computes \( H_n \) for a given \( n \in \mathbb{N} \). You only need a single loop running through the numbers 1 up to \( n \), adding their reciprocals. Just as well you can make your loop run from \( n \) down to 1 and sum up the reciprocals. Why not, that should not make any difference, right? Let us try both variants and see what we get. The program harmonic.cpp shown below computes the two sums and outputs them.
# Program 13: ../progs/lecture/harmonic.cpp

Think for a second and recall why it is important to not write \( \frac{1}{i} \) in line 16 and line 21. Now let us have a look at an execution of the program. On the platform of the authors, the following happens.

Compute \( H_n \) for \( n =? \) 10000000
Forward sum = 15.4037
Backward sum = 16.686

The results differ significantly. The difference becomes even more apparent when we try larger inputs.

Compute \( H_n \) for \( n =? \) 100000000
Forward sum = 15.4037
Backward sum = 18.8079

Notice that the forward sum did not change, which cannot be correct. Using the
approximation
\[ \frac{1}{2(n+1)} < H_n - \ln n - \gamma < \frac{1}{2n}, \]
where \( \gamma = 0.57721666\ldots \) is the Euler-Mascheroni constant, we get \( H_n \approx 18.998 \) for \( n = 10^8 \). That is, the backward sum provides a much better approximation of \( H_n \).

Why does the forward sum behave so badly? The reason is simple: As the larger summands are added up first, the intermediate value of the sum to be computed grows (comparatively) fast. At some point, the size difference between the partial sum and the summand \( \frac{1}{n} \) to be added is so large that the addition does not change the partial sum anymore, just like in \( 2^n + 1 \approx 2^n \). Thus, regardless of how many more summands are added to it, the sum stays the same.

In contrast, the backward sum starts to add up the small summands first. Therefore, the value of the partial sum grows (comparatively) slowly, allowing the small summands to contribute. The summands treated in the end of the summation have still a good chance to influence the significand of the partial sum, since they are (comparatively) large.

The phenomenon just observed leads us to our second guideline.

Floating Point Arithmetic Guideline 2: Avoid adding two numbers that considerably differ in size.

Cancellation. Consider the quadratic equation
\[ ax^2 + bx + c = 0, \quad a \neq 0. \]
It is well known that its two roots are given by
\[ r_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \]
In a program that computes these roots, we might therefore want to compute the value \( d = b^2 - 4ac \) of the discriminant. If \( b^2 \) and \( 4ac \) are representable as floating point numbers with precision \( p \), our previous error estimates guarantee that the result \( \hat{d} \) of the final subtraction has small relative error: \(|d - \hat{d}| \leq 2^{-p}|d|\). This means, even if \( d \) is close to zero, \( \hat{d} \) will be away from \( d \) by much less than the distance of \( d \) to zero.

The problem arises if the numbers \( b^2 \) and/or \( 4ac \) are not representable as floating point numbers, in which case errors are made in computing them. Assume \( b = 2^p, a = 2^{p-1} - 1, c = 2^{p-1} + 1 \) (all these numbers are exactly representable). Then the exact value of \( d \) is 4. The value \( b^2 = 2^{2p} \) is a representable floating point number, but \( 4ac = 2^{2p} - 4 \) is not, since this number has \( 2p - 2 \) significant digits (all of them equal to 1) in its binary expansion. The nearest floating point number is obtained by rounding up (adding 4), and
after the (error-free) subtraction, we get \( \hat{d} = 0 \). The relative error of this computation is therefore 1 instead of \( 2^{-p} \).

The reason is that in subtracting two numbers that are almost equal, the more significant digits cancel each other. If, on the other hand, the remaining less significant digits already carry some errors from previous computations, the subtraction hugely amplifies these errors: the cancellation promotes the previously less significant digits to much more significant digits of the result.

Again, the example we gave here is artificial, but be assured that cancellation happens in practice. Even in the quadratic equation example, it might be that the equations that come up in an application have the property that their discriminant \( b^2 - 4ac \) is much smaller than \( a, b \) and \( c \) themselves. In this case, cancellation will happen.

The discussion can be summarized in form of a third guideline.

Floating Point Arithmetic Guideline 3: Avoid subtracting two numbers of almost equal size, if these numbers are results of other floating point computations.

2.5.8 Details

Other floating point number systems. The IEEE standard 754 defines two more floating point number systems, single-extended precision (\( p = 32 \)), and double-extended precision (\( p = 64 \)), and some platforms offer implementations of these types. There is also the IEEE standard 854 that allows base \( \beta = 10 \), for obvious reasons: the decimal format is the one in which we think about numbers, and in which we usually represent numbers. In particular, a base-10 system has no holes in the value range at decimal fractional numbers like 1.1 and 0.1.

IEEE compliance. While on most platforms, the types float and double correspond to the single and double precision floating point numbers of the IEEE standard 754, this correspondence is usually not one-to-one. For example, if you are trying to reproduce the cancellation example we gave, you might write

```cpp
const float b = 16777216.0f; // 2^24
const float a = 8388607.0f; // 2^23 - 1
const float c = 8388609.0f; // 2^23 + 1
std::cout << b * b - 4.0f * a * c << "\n";
```

and expect to get the predicted wrong result 0. But it may easily happen that you get the correct result 4, even though your platform claims to follow the IEEE standard 754. The most likely reason is that the platform internally uses a register with more bits to perform the computation. While this seems like a good idea in general, it can be fatal for a program whose functionality critically relies on the IEEE standard 754.

You will most likely see the cancellation effect in the following seemingly equivalent variant of the above code.
float b = 16777216.0f;  // 2^24
float a = 8388607.0f;  // 2^23 - 1
float c = 8388609.0f;  // 2^23 + 1

float bb = b * b;
float ac4 = 4.0f * a * c;

std::cout << bb - ac4 << "\n";

Here, the results of the intermediate computations are written back to float variables, probably resulting in the expected rounding of 4ac. Then the final subtraction reveals the cancellation effect. Unless, of course, the compiler decides to keep the variable ac4 in a register with more precision. For this reason, you can typically provide a compiler option to make sure that floating point numbers are not kept in registers.

What is the morale of this? You usually cannot fully trust the IEEE compliance of a platform, and it is neither easy nor worthwhile to predict how floating point numbers exactly behave on a specific platform. It is more important for you to know and understand floating point number systems in general, along with their limitations. This knowledge will allow you to identify and work around problems that might come up on specific platforms.

The type long double. The C++ standard prescribes another fundamental floating point number type called long double. Its literals end with the letter l or L, and it is guaranteed that the value range of double is contained in the value range of long double. Despite this, the conversion from double to long double is not defined to be a promotion by the C++ standard.

While float and double usually correspond to single and double precision of the IEEE standard 754, there is no such default choice for long double. In practice, long double might simply be a synonym for double, but it might also be something else. On the platform used by the authors, for example, long double corresponds to the normalized floating point number system F'(2, 64, -16382, 16384)—this is exactly the double-extended precision of the IEEE standard 754.

Numeric limits. If you want to know the parameters of the floating point number systems behind float, double and long double on your platform, you can employ the numeric_limits we have used before in the program limits.cpp in Section 2.2.5. Here are the relevant expressions together with their meanings, shown for the type float.

<table>
<thead>
<tr>
<th>expression (of type int)</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>std::numeric_limits&lt;float&gt;::radix</td>
<td>β</td>
</tr>
<tr>
<td>std::numeric_limits&lt;float&gt;::digits</td>
<td>p</td>
</tr>
<tr>
<td>std::numeric_limits&lt;float&gt;::min_exponent</td>
<td>e_{min} + 1</td>
</tr>
<tr>
<td>std::numeric_limits&lt;float&gt;::max_exponent</td>
<td>e_{max} + 1</td>
</tr>
</tbody>
</table>
We remark that std::numeric_limits<float>::min() does not give the smallest float value (because of the sign bit, this smallest value is simply the negative of the largest value), but the smallest normalized positive value.

Special numbers. We have mentioned that the floating point systems prescribed by the IEEE standard 754 contain some special numbers; their encoding uses exponent values that do not occur in normalized numbers.

On the one hand, there are the denormalized numbers of the form

$$\pm d_0.d_1\ldots d_{p-1} \cdot \beta^{e_{\min}},$$

with $d_0 = 0$. A denormalized number has smaller absolute value than any normalized number. In particular, 0 is a denormalized number.

The other special numbers cannot really be called numbers. There are values representing $+\infty$ and $-\infty$, and they are returned by overflowing operations. Then there are several values called NaNs (for "not a number") that are returned by operations with undefined result, like taking the square root of a negative number. The idea behind these values is to provide more flexibility in dealing with exceptional situations. Instead of simply aborting the program when some operation fails, it makes sense to return an exceptional value. The caller of the operation can then decide how to deal with the situation.

2.5.9 Goals

Dispositional. At this point, you should ...

1) know the floating point number types float and double, and that they are more general than the integral types;

2) understand the concept of a floating point number system, and in particular its advantages over a fixed point number system;

3) know that the IEEE standard 754 describes specific floating point number systems used as models for float and double on many platforms;

4) know the three Floating Point Arithmetic Guidelines;

5) be aware that computations involving the types float and double may deliver inexact results, mostly due to holes in the value range.

Operational. In particular, you should be able to ...

(G1) evaluate expressions involving the arithmetic types int, unsigned int, float and double;

(G2) compute the binary representation of a given real number;
compute the floating point number nearest to a given real number, with respect to a finite floating point number system;
work with a given floating point number system;
recognize usage of floating point numbers that violates any of the three Floating Point Arithmetic Guidelines;
write programs that perform computations with floating point numbers.

2.5.10 Exercises

Exercise 56 For every expression in the following list, determine its type, its value, and whether it is an rvalue or an lvalue. In each of the expressions, the variable $x$ is of type int and has value 1.

a) $1 + \text{true} == 2$

b) $3 \% 2 + 1 * 4$

c) $x = 10 / 2 / 5 / 2$

d) $x / 2.0$

e) $1 + x++$

f) $++x$

Exercise 57 For every expression in the following list, determine its type and its value. We assume a floating point representation according to IEEE 754, that is, float corresponds to $F'(2, 24, -126, 127)$ and double to $F'(2, 53, -1022, 1023)$. We also assume that 32 bits are used to represent int values.

a) $2e2-3e3f>-23.0$

b) $-7+7.5$

c) $1.0f/2+1/3+1/4$

d) $\text{true}||\text{false}&&\text{false}$

e) $1u-2u<0$

f) $1+2*3+4$

g) $\text{int}(8.5)-\text{int}(7.6)$

h) $100*1.1==110$

i) $10*11.0==110$
2.5. FLOATING POINT NUMBERS

j) \(4 + 12 / 4.0 / 2\)

**Exercise 58** Evaluate the following expressions step-by-step, according to the conversion rules of mixed expressions. We assume a floating point representation according to IEEE 754, that is, float corresponds to \( F'(2, 24, -126, 127) \) and double corresponds to \( F'(2, 53, -1022, 1023) \). We also assume that 32 bits are used to represent int values.

\[
\begin{align*}
a) & \quad 6 / 4 * 2.0f - 3 \\
b) & \quad 2 + 15.0e7f - 3 / 2.0 * 1.0e8 \\
c) & \quad 392593 * 2735.0f - 8192 * 131072 + 1.0 \\
d) & \quad 16 * (0.2f + 262144 - 262144.0)
\end{align*}
\]

**Exercise 59** Compute the binary expansions of the following decimal numbers.

\[
\begin{align*}
a) & \quad 0.25 \\
b) & \quad 1.52 \\
c) & \quad 1.3 \\
d) & \quad 11.1
\end{align*}
\]

**Exercise 60** For the numbers in Exercise 59, compute nearest floating point numbers in the systems \( F'(2, 5, -1, 2) \) and \( F(2, 5, -1, 2) \).

**Exercise 61** What are the largest and smallest positive normalized single and double precision floating point numbers, according to the IEEE standard 754?

**Exercise 62** How many floating point numbers do the systems \( F'(\beta, p, e_{\min}, e_{\max}) \) and \( F(\beta, p, e_{\min}, e_{\max}) \) contain?

**Exercise 63** Compute the value of the variable \( d \) after the declaration statement

\[
\text{float } d = 0.1;
\]

*Assume the IEEE standard 754.*

**Exercise 64** What is the problem with the following loop (assuming the IEEE standard 754)?

\[
\begin{align*}
\text{for (float } i = 0.1f; i != 1.0f; i += 0.1f) \\
& \quad \text{std::cout } \ll i \ll "\n";
\end{align*}
\]

**Exercise 65** What is the problem with the following loop (assuming the IEEE standard 754)?

\[
\begin{align*}
\text{for (float } i = 0.0f; i < 100000000.0f; ++i) \\
& \quad \text{std::cout } \ll i \ll "\n";
\end{align*}
\]
Exercise 66 Write a program that outputs for a given decimal input number $x$, $0 < x < 2$, its normalized float value on your platform. The output should contain the (binary) digits of the significand, starting with 1, and the (decimal) exponent. You may assume that the floating point number system underlying the type float has base $\beta = 2$.

Exercise 67 Write a program that tests whether a given value of type double is actually an integer, and test the program with various inputs like 0.5, 1, 1234567890, 1234567890.2. Simply converting to a value of type int and checking whether this changes the value does not work in general, since the given value might be an integer outside the value range of int. You may assume that the floating point number system underlying the type double has base $\beta = 2$.

Exercise 68 The number $\pi$ can be defined through various infinite sums. Here are two of them.

\[
\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots \\
\frac{\pi}{2} = 1 + \frac{1}{3} + \frac{1.2}{3.5} + \frac{1.2.3}{3.5.7} + \cdots
\]

Write a program for computing an approximation of $\pi$, based on these formulas. Which formula is better for that purpose?

Exercise 69 The function $\sin(x)$ can be defined through the following power series.

\[
\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots
\]

a) Implement and test—based on the first 20 terms of this power series—a function

```c
// POST: returns an approximation of sin(x)
double sinus (double x);
```

b) What happens if the argument value is very large, $x = 10,000$, say? Does your function still yield reasonable results, then? If not, do you see a way to fix this?

Exercise 70 There is a well-known iterative procedure (the Babylonian method) for computing the square root of a positive real number $s$. Starting from any value $x_0 > 0$, we compute a sequence $x_0, x_1, x_2, \ldots$ of values according to the formula

\[
x_n = \frac{1}{2} \left( x_{n-1} + \frac{s}{x_{n-1}} \right).
\]
It can be shown that
\[ \lim_{n \to \infty} x_n = \sqrt{s}. \]

Write a program `babylonian.cpp` that reads in the number \( s \) and computes an approximation of \( \sqrt{s} \) using the Babylonian method. To be concrete, the program should output the first number \( x_i \) such that
\[ |x_i^2 - s| < 0.001. \]

**Exercise 71** Write a program `fpsys.cpp` to visualize a normalized floating point number system \( \mathcal{F}^*(2, p, e_{\min}, e_{\max}) \). The program should read the parameters \( p, e_{\min}, \) and \( e_{\max} \) as inputs and for each positive number \( x \) from \( \mathcal{F}^*(2, p, e_{\min}, e_{\max}) \) draw a circle of radius \( x \) around the origin. Use the library `libwindow` that is available at the course homepage to create graphical output. Use the program to verify the numbers you computed in Exercise 62.

**Exercise 72** We have seen that the decimal number 0.1 has no finite representation in a binary floating-point number system (\( \beta = 2 \)). Mr. X. M. Plestudent claims that this is due to \( \beta < 10 \). He suggests to work with a hexadecimal system (\( \beta = 16 \)) and argues that in such a system, 0.1 does have a finite representation. Is Mr. Plestudent right or not?

Somewhat more formally, is there a natural number \( p \) and numbers \( d_1, \ldots, d_p \) with \( d_i \in \{0, 1, \ldots, 15\} \) for all \( i \), such that
\[ \frac{1}{10} = \sum_{i=1}^{p} d_i 16^{-i} = "0.d_1 \ldots d_p" \]
holds?

**Exercise 73** We have seen that there are decimal numbers without a finite binary representation (such as 1.1 and 0.1). Conversely, every (fractional) binary number does have a finite decimal representation, a fact that may be somewhat surprising at first sight. Prove this fact!

More formally, given a number \( b \) of the form
\[ b = \sum_{i=1}^{k} b_i 2^{-i}, \quad b_1, b_2, \ldots, b_k \in \{0, 1\}, \]
prove that there is a natural number \( \ell \) such that \( b \) can be written as an \( \ell \)-digit decimal number
\[ b = \sum_{i=1}^{\ell} d_i 10^{-i}, \quad d_1, d_2, \ldots, d_\ell \in \{0, 1, \ldots, 9\}. \]
2.5.11 Challenges

Exercise 74 We have seen that decimal numbers do not necessarily have finite binary representations (examples are the decimal numbers 1.1 and 0.1). Vice versa, binary numbers do have finite decimal representations (Exercise 73). And Exercise 72 asks whether every decimal number has a finite hexadecimal representation. The goal of this challenge is to understand the general picture.

Let $\beta \geq 2$ and $\gamma \geq 2$ be two natural numbers. We say that $\gamma$ refines $\beta$ if for every $p, e_{\min}, e_{\max}, q, f_{\min}, f_{\max}$, there are $q, f_{\min}, f_{\max}$ such that

$$F(\beta, p, e_{\min}, e_{\max}) \subseteq F(\gamma, q, f_{\min}, f_{\max}).$$

In other words, every floating-point number system to the base $\beta$ is contained in some floating-point number system to the base $\gamma$.

In this language, the result of Section 2.5.5 is that 2 does not refine 10. Exercise 73 implies that 10 refines 2, and Exercise 72 asks whether 16 refines 10.

Here is the challenge: characterize the pairs $(\beta, \gamma)$ for which $\gamma$ refines $\beta$!

Exercise 75 The Mandelbrot set is a subset of the complex plane that became popular through its fractal shape and the beautiful drawings of it. Below you see the set’s main cardioid and a detail of it at much higher zoom scale.

The Mandelbrot set is defined as follows. For $c \in \mathbb{C}$, we consider the sequence $z_0(c), z_1(c), \ldots$ of complex numbers given by $z_0(c) = 0$ and

$$z_n(c) = z_{n-1}(c)^2 + c, \quad n > 0.$$

There are two cases: either $|z_n(c)| \leq 2$ for all $n$ (this obviously happens for example if $c = 0$), or $|z_n(c)| > 2$ for some $n$ (this obviously happens for example if $|c| > 2$). The Mandelbrot set consists of all $c$ for which we are in the first case. It follows that the Mandelbrot set contains 0 and is contained in a disk of radius 2 around 0 in the complex plane.

Write a program that draws (an approximation of) the Mandelbrot set, restricted to a rectangular subset of the complex plane. It should be possible to zoom in, meaning that the rectangular subset becomes smaller, and more details become visible in
the drawing window. Obviously, you can’t process all infinitely many complex numbers \(c\) in the rectangle, and for given \(c\), you cannot really check whether \(|z_n(c)| \leq 2\) for all \(n\), so it is necessary to discretize the rectangle into pixels, and to establish some upper bound \(N\) on the number of iterations. If \(|z_n(c)| \leq 2\) for all \(n \leq N\), you may simply assume that \(c\) is in the Mandelbrot set. Per se, there is no guarantee that the resulting drawing is even close to the Mandelbrot set (especially at finer level of detail), but for the sake of obtaining nice pictures, we can generously gloss over this issue.

**Hint:** You may use the libwindow library to produce the drawing. The example program in its documentation should give you an idea how this can be done.

**Exercise 76** The following email was sent to a mailing list for users of the software library CGAL.

Hi all,

This should be a very easy question.

When I check if the points (0.14, 0.22), (0.15, 0.21) and (0.19, 0.17) are collinear, using CGAL::orientation, it returns CGAL::LEFT_TURN, which is false, because those points are in fact collinear.

However, if I do the same with the points (14, 22), (15, 21) and (19, 17) I get the correct answer: CGAL::COLLINEAR.

a) Find out what this email is about; in particular, what is CGAL, what is the orientation of a point triple, what is CGAL::orientation, what does “collinear” mean, and why is the writer of the email surprised about the observed behavior?

b) Draft an answer to this email that explains the observations of the CGAL user that wrote it.
2.6 Arrays and pointers

As all real programmers know, the only useful data structure is the array.

Ed Post, Real Programmers don't use Pascal (1983)

Reading into an array without making a "silly error" is beyond the ability of complete novices - by the time you get that right, you are no longer a complete novice.

Bjarne Stroustrup, C++ Style and Technique FAQ

This section introduces arrays as containers for sequences of objects of the same type, with random access to individual members of the sequence. An array is the most primitive but at the same time a very efficient container for storing, processing, and iterating over large amounts of data. You will also learn about pointers as explicit object addresses and about their close relationship with arrays. While the C++ standard library contains less primitive and generally better alternatives, the concepts behind arrays and pointers are of fundamental importance.

In Section 2.4 on control statements, we have learned about the concept of iteration. For example, we can now iterate over the sequence of numbers 1, 2, \ldots, n and perform some operations like adding up all the numbers, or identifying the prime numbers among them. Similarly, we can iterate over the odd numbers, the powers of two, etc.

In real applications, however, we often have to process (and in particular iterate over) sequences of data. For example, if you want to identify the movie theaters in town that show your desired movie tonight, you have to iterate over the sequence of movie theater repertoires. These repertoires must be stored somewhere, and there must be a way to inspect them in turn. In C++, we can deal with such tasks by using arrays.

2.6.1 Array types

An array of length \( n \) aggregates \( n \) objects of the same type \( T \) into a sequence. To access one of the aggregated objects (the elements), we use its index or subscript (position) in the sequence. All these length-\( n \) sequences form an array type whose value range corresponds to the mathematical type \( T^n \). In the computer's main memory, an array occupies a contiguous part, with the elements stored side-by-side (see Figure 7).

Let us start by showing an array in action: Eratosthenes' Sieve is a fast method for computing all prime numbers smaller than a given number \( n \), based on crossing out the
numbers that are not prime. It works like this: you write down the sequence of numbers between 2 and \( n - 1 \). Starting from 2, you always go to the next number not crossed out yet, report it as prime, and then cross out all its proper multiples.

Let's not dwell on the correctness of this method but go right to the implementation. If you think about it for a minute, the major question is this: how do we cross out numbers?

The following program uses an array type variable `crossed_out` for the list, where every value `crossed_out[i]` is of type `bool` and represents the (changing) information whether the number \( i \) has already been crossed out or not. Array indices always start from 0, so in order to get to index \( n - 1 \), we need an array of length \( n \). The program runs Eratosthenes' Sieve for \( n = 1,000 \).

```cpp
1 // Program: eratosthenes.cpp
2 // Calculate prime numbers in \{2,...,n-1\} using
3 // Eratosthenes' sieve.
4
5 #include <iostream>
6
7 int main()
8 {
9     const unsigned int n = 1000;
10
11     // definition and initialization: provides us with
12     // Booleans crossed_out[0],..., crossed_out[n-1]
13     bool crossed_out[n];
14     for (unsigned int i = 0; i < n; ++i)
15         crossed_out[i] = false;
16
17     // computation and output
18     std::cout << "Prime numbers in \{2,...,\} \n";
19     for (unsigned int i = 2; i < n; ++i)
20         if (!crossed_out[i]) {
21             // i is prime
22             std::cout << i << " 
";
23             // cross out all proper multiples of i
24             for (unsigned int m = 2*i; m < n; m += i)
25                 crossed_out[m] = true;
26         }
27     std::cout << "\n";
28
29     return 0;
30 }
```

Program 14: ../progs/lecture/eratosthenes.cpp
Definition. An array variable (or simply array) \( a \) with \( n > 0 \) elements of underlying type \( T \) is defined through the following declaration.

\[
T a[expr]
\]

Here, \( expr \) must be a constant expression of integral type whose value is \( n \). For example, literals like 1000, arithmetic expressions over literals (like 1+1), and constants (Section 2.1.9) are constant expressions; all of them have the property that their value is known at compile time. This allows the compiler to figure out how much memory the array variable needs.

The type of \( a \) is \( T^n \), but we put this in double quotes here (only to omit them later). The reason is that \( T^n \) is not the official name: we can't write \( \text{int}[5] \ a \), for example, to declare an array \( a \) of type \( \text{int}[5] \).

The value range of \( T^n \) is \( T^n \), the set of all sequences \( (t_1, t_2, \ldots, t_n) \) with all \( t_i \) being of type \( T \).

The fact that the array length must be known at compile time clearly limits the usefulness of array variables. For example, this limitation does not allow us to write a version of Eratosthenes' sieve in which the number \( n \) is read from the input. But we will shortly see how this restriction can be overcome—for the time being, let's simply live with it.

### 2.6.2 Initializing arrays

The definition of an array with underlying fundamental type does not initialize the values of the array elements. We can assign values to the elements afterwards (like we do it in Program 14), but we can also provide the values directly, as in the following declaration statement.

\[
\text{int } a[5] = \{4, 3, 5, 2, 1\};
\]

Since the number of array elements can be deduced from the length of the initializer list, we can also write

\[
\text{int } a[] = \{4, 3, 5, 2, 1\};
\]

The declaration \( \text{int } a[] \) without any initialization is invalid, though, since it does not fully determine the type of \( a \). We say that \( a \) has incomplete type in this case.

### 2.6.3 Random access to elements

The most common and useful way of accessing and modifying the elements of an array is by random access. If \( expr \) is of integral type and has value \( i \), the lvalue

\[
a[expr]
\]
is of the type underlying the array $a$ and refers to the $i$-th element (counting from 0) of $a$. The number $i$ is called the index or subscript of the element. If $n$ is the length of $a$, the index $i$ must satisfy $0 \leq i < n$. The operator $[\cdot]$ is called the subscript operator.

The somewhat strange declaration format of an array is motivated by the subscript operator. Indeed, the declaration

$$T a[\text{expr}]$$

can be read as “$a[\text{expr}]$ is of type $T$”. In this sense, it is an implicit definition of $a$’s type.

Watch out! You as a programmer are responsible for making sure that a given array index $i$ indeed satisfies $0 \leq i < n$, where $n$ is the length of the array. Indices that are not in this range are called out of bound. Unless your compiler offers specific debugging facilities, the usage of out-of-bound indices in the subscript operator is not checked at runtime and leads to undefined behavior of the program.

We have already discussed the term random access in connection with the computer’s main memory (Section 1.2.3); random access means that every array element can be accessed in the same uniform way, and with (almost) the same access time, no matter what its index is. Evaluating the expression $a[0]$ is as fast as evaluating $a[10000]$. In contrast, the thick pile of pending invoices, bank transfers and various other papers on your desk does not support random access: the time to find an item is roughly proportional to its depth within the pile.

In fact, random access in an array directly reduces to random access in the computer’s main memory, since an array always occupies a contiguous set of memory cells, see Figure 7.

![Figure 7: An array occupies a contiguous part of the main memory. Every element in turn occupies $s$ memory cells, where $s$ is the number of memory cells required to store a single value of the underlying type $T$.](image)

To access the element of index $i$ in the array $a$, a simple computation with addresses therefore suffices. If $p$ is the address (position) where the first element of a “starts”, and $s$ is the number of memory cells that a single value of the underlying type $T$ occupies, then the element of index $i$ starts at the memory cell whose address is $p + si$, see Figure 8.

2.6.4 Arrays are primitive

Array types are exceptional in C++. The following code fragment illustrates this:
CHAPTER 2. FOUNDATIONS

\[
p \quad p + s \quad p + 2s \quad \cdots \quad p + (n - 1)s
\]

\[
\begin{array}{ccc}
0 & 1 & 2 \\
\hline
\hline
\end{array}
\]

\[
\begin{array}{c}
\text{index} \\
\hline
n - 1 \\
\end{array}
\]

Figure 8: The array element of index \(i\) starts at the address \(p + si\).

\[
\text{int } a[5] = \{4,3,5,2,1\}; // \text{array of type int[5]} \\
\text{int } b[5]; \\
\text{b = a;} \quad // \text{error: we cannot assign to an array}
\]

We also cannot initialize an array from another array. Why is this? Arrays are a dead hand from the programming language C, and the design of arrays in C is quite primitive. C was designed to compete with machine language in efficiency, and this didn’t leave room for luxury. We can of course copy one array into another manually via a simple loop, and early C programmers were not yet spoiled enough to complain about this bit of extra work.

When C++ was developed much later, one design goal was to have C as a subset. As a consequence, arrays are still around in C++, in their original form. Internally, an array is represented by the address of its first memory cell, and by its memory size in bytes, see Section 1.2.3. In general, if \(\text{obj}\) is any object of any type, the expression \(\text{sizeof(}\text{obj})\) returns the number of memory cells used by its argument, under the agreement that one memory cell is required to store an object of type \(\text{char}\).

The argument of \(\text{sizeof}\) can also be the name of a type, in which case the number of memory cells occupied by one object of this type is returned (the number \(s\) in Figure 7). Thus, to find out how many elements an array \(a\) of underlying type \(\text{int}\) has, we may use the somewhat clumsy expression \(\text{sizeof(}\text{a}) / \text{sizeof(}\text{int})\).

2.6.5 Iteration over a container

Let’s take a step back, forget about the technicalities of arrays for a moment, and go for a bigger picture.

We have already indicated in the introduction to this section that the process of iterating over a sequence of data is ubiquitous. Typically, the data are stored in some
container, and we need to perform a certain operation for all elements in the container. In general, a container is an object that can store other objects (its elements), and that offers some ways of accessing these elements. The only “hard” requirement here is that a container must offer the possibility of iterating over all its elements. In this informal sense, an array is indeed a container, since the random access functionality can be used to iterate over the elements.

Iteration by random access. Let’s get back to arrays. Iterating over an array of length $n$ can be done by random access like in lines 14–15 of Program 14. We have seen that the random access functionality of arrays is internally based on address arithmetic. During the iteration, the following sequence of addresses is computed: $p, p + s, p + 2s, \ldots, p + (n - 1)s$, where $p$ and $s$ have the usual meanings.

This requires one multiplication and one addition for every address except the first. But if you think about it, the multiplication only comes in because we compute each address from scratch, independently from the previous ones. In fact, the same set of addresses could more efficiently and more naturally be computed by starting with $p$ and repeatedly adding $s$ (“going to the next element”).

Using random access, we can simulate array iteration, but we are missing the operation of “going to the next element”; only this operation makes iteration over a container natural and efficient. The following analogy illustrates the point: you can of course read a book by starting with page 1, then closing the book, opening it again on pages 2 – 3, closing it, opening it on pages 4 – 5, etc. But unless you’re somewhat eccentric, you probably prefer to just turn the pages in between.

Iteration by pointers. Arrays offer natural and efficient iteration through pointers. Pointer values can be thought of as actual addresses, and they allow operations like “adding $s$” in order to go to the next element in the array. Here is how we could equivalently write the iteration in lines 14–15 of Program 14 with pointers.

```c
bool* const begin = crossed_out; // pointer to first element
bool* const end = crossed_out + n; // past-the-end pointer
// in the loop, pointer p successively points to all elements
for (bool* p = begin; p != end; ++p)
    *p = false; // *p is the element pointed to by p
```

Admittedly, this looks more complicated at first sight than the random access version, but we’ll explain what’s going on in detail in the next sections. In terms of Figure 8, we have replaced iteration by index with iteration by address.

2.6.6 Pointer types and functionality

For every type $T$ the corresponding pointer type is

$T*$
We call $T$ the underlying type of $T^*$. An expression of type $T^*$ is called a pointer (to $T$).

The value of a pointer to $T$ is the address of (the first memory cell of) an object of type $T$. We call this the object pointed to by the pointer.

We can visualize a pointer $p$ as an arrow pointing to a cell in the computer's main memory—the cell where the object pointed to starts, see Figure 9.

![Figure 9: A pointer to $T$ represents the address of an object of type $T$ in the computer's main memory.](image)

Figure 9: A pointer to $T$ represents the address of an object of type $T$ in the computer's main memory.

Initialization, the assignment operator $=$, and the comparison operators $==$ and $!=$ are defined for every pointer type $T^*$. The latter simply test whether the addresses in question are the same or not.

Initialization and assignment copy the value (as usual), which in this case means to copy an address; thus, if $j$ points to some object, the assignment $i = j$ has the effect that $i$ now also points to this object. The object itself is not copied. We remark that pointer initialization and assignment require the types of both operands to be exactly the same—implicit conversions don't work. If you think about it, this is clear. Imagine that the variable $i$ is of type int*, and that you could write

```c
double* j = i
```

Since double objects usually require more memory cells than int objects, $j$ would now be a pointer to a double object that includes memory cells originally not belonging to $i$. This can hardly be called a “conversion”. In fact, since we only copy an address, there cannot be any physical conversion of the stored value, even if the memory requirements of the two types happen to be the same.

The address operator. We can obtain a pointer to a given object by applying the unary address operator to an lvalue that refers to the object. If the lvalue is of type $T$, then the result is an rvalue of type $T^*$. The syntax of an address operator call is

```c
&lvalue
```

In the following code fragment we use the address operator to initialize a variable iptr of type int* with the address of an object of type int named i.

```c
int i = 5;
int* iptr = &i; // iptr initialized with the address of i
```
### 2.6. ARRAYS AND POINTERS

The **dereference operator**. From a pointer, we can get back to the object pointed to through *dereferencing* or *indirection*. The unary dereference operator `*` applied to an rvalue of pointer type yields an lvalue referring to the object pointed to. If the rvalue is of type `T*`, then the result is of type `T`. The syntax of a dereference operator call is `*rvalue`

Following up on our previous code fragment, we can therefore write

```c
int i = 5;
int* iptr = &i; // iptr initialized with the address of i
int j = *iptr; // j == 5
```

The naming scheme of pointer types is motivated by the dereference operator. The declaration `T* p` can also be read (and in fact legally be written; we don’t do this, though) as `T *p`

The second version implicitly defines the type of `p` by saying that `*p` is of type `T`. This is the same kind of implicit definition that we already know from array declarations.

Figure 10 illustrates address and dereference operator.

![Figure 10: The address operator (left) and its inverse, the dereference operator (right)](image)

**The null pointer.** For every pointer type there is a value distinguishable from any other pointer value. This value is called the *null pointer value*. The integer value `0` can be converted to every pointer type, and the value after conversion is the null pointer value. In the declaration `int* iptr = 0`, for example, the variable `iptr` gets initialized with...
the null pointer value. We also say that `iptr` is a *null pointer*. The null pointer value must not be dereferenced, since it does not correspond to any existing address.

Using the null pointer value is the safe way of indicating that there is no object (yet) to point to. The alternative of leaving the pointer uninitialized is bad: there is no way of testing whether a pointer that is not a null pointer holds the address of a legitimate object, or whether it holds some "random" address resulting from leaving the pointer uninitialized.

In the latter case, dereferencing the pointer usually crashes the program. Consider this code:

```c
int * iptr;  // uninitialized pointer
int j = *iptr;  // trouble!
```

After its declaration, the pointer `iptr` has undefined value, which in practice means that it may correspond to an arbitrary address in memory; dereferencing it means to access the memory content at this address. In general, this address will not belong to the part of memory to which the program has access; the operating system will then deny access to it and terminate the program with a *segmentation fault*.

### 2.6.7 Array-to-pointer conversion

Any array of type `T[n]` can implicitly be converted to type `T*`. The resulting value is the address of the first element of the array. For example, we can write

```c
int a[5];
int* begin = a;  // begin points to a[0]
```

The declaration

```c
int* begin = &a[0];  // address of the first element
```

is equivalent as far as the resulting value of `begin` is concerned, but there is a subtle difference: the latter declaration evaluates `a[0]`, while the former does not.

The pointer-style replacement code for the loop in lines 14–15 of Program 14 that we have presented at the end of Section 2.6.5 makes use of array-to-pointer conversion in the first line:

```c
bool* const begin = crossed_out;  // pointer to first element
```

The array-to-pointer conversion is trivial on the machine side. For this, we recall from our earlier discussion in Section 2.6.4 that in C and therefore also in C++, the address of the first element is part of the array's internal representation. It is important to understand, though, that the *length* of the array gets lost during array-to-pointer conversion: the resulting pointer is just a pointer, and nothing else.

Array-to-pointer conversion automatically takes place when an array appears in an expression. The only exception are expressions of the form `sizeof(a)`, where `a` is an array. Bjarne Stroustrup, the designer of C++, illustrates this by saying that *the name of an array converts to a pointer to its first element at the slightest provocation.*
In still other words, there are no operations on arrays, except the length computations through `sizeof`: everything that we conceptually do with an array is in reality done with a pointer to its first element; this in particular applies to the random access operation, see the paragraph called “Pointer subscripting, or the truth about random access” in Section 2.6.9 below.

### 2.6.8 Pointers and const

Let us discuss the strange placement of the `const` keyword in the line

```c
bool* const begin = crossed_out; // pointer to first element
```

This declares the pointer variable `begin` to be a *constant* pointer (Section 2.1.9) whose value is the address of the first element of the array `crossed_out`. So far, we have declared constants by putting `const` in front of the declaration. With pointers, this can also be done but it has a different meaning. The declaration

```c
const bool* begin = crossed_out;
```

would make `begin` a (non-constant) pointer with underlying type `const bool`. This would mean that not the pointer `begin` itself, but the *object* pointed to by `begin` is constant. In this case, `begin` is called a *const pointer*. We don't want this meaning, as it would not allow us to write anything into the array `crossed_out`. Here is why.

If the object pointed to by `begin` is constant, a direct assignment

```c
*begin = false;
```

that tries to set the first element of `crossed_out` to `false` would lead to an error message. In our pointer-style initialization loop on page 139, we are doing the assignments with a running pointer `p` that initially has value `begin`. Since `p` has underlying type `bool` and not `const bool`, shouldn't it be possible to assign `false` to `*p`? Yes, but in code such as

```c
const bool* begin = crossed_out;
...
for (bool* p = begin; ... )
...
```

the compiler will already reject the loop's *init-statement* `bool* p = begin;` with an error message telling us that it cannot convert from type `const bool*` to `bool*`, since this would make the object pointed to by `begin` non-constant through the backdoor. (The conversion in the other direction is no problem, since it only "adds constness").

Const-qualified types should be used as underlying types of arrays (and pointers to array elements) *if and only if* the program does not intend to modify the values of the array elements. In the program `eratosthenes.cpp`, we *do* want to modify the array, but the pointers `begin` and `end` that tell us “where the array lives” won’t change. Following the Const Guideline (Page 2.1.9), we thus use the keyword `const` here to make these pointers themselves constant and obtain const-correct code.
What exactly is constant? With a pointer \( p \) of type \( \text{const } T^* \), we cannot achieve “absolute constness” of the object pointed to by \( p \). We only achieve that the object can neither directly nor indirectly be modified through the pointer \( p \). But we could modify it through another pointer. Consider the code fragment

```c
const bool* begin1 = crossed_out;
bool* begin2 = crossed_out;
*begin1 = false;            // error
*begin2 = false;            // ok
```

Here, \( \text{begin1} \) and \( \text{begin2} \) are two pointers to the same object, namely the first element of the array \( \text{crossed_out} \). With respect to the pointer \( \text{begin1} \), this element is constant, but with respect to \( \text{begin2} \), it is not.

### 2.6.9 Pointer arithmetic

In order to understand why the code fragment

```c
bool* const begin = crossed_out;       // pointer to first element
bool* const end = crossed_out + n;     // past-the-end pointer
// in the loop, pointer \( p \) successively points to all elements
for (bool* p = begin; p != end; ++p)
    *p = false;       // *p is the element pointed to by \( p \)
```

indeed sets all elements of the array \( \text{crossed_out} \) to \( \text{false} \), we have to understand pointer arithmetic, the art of computing with addresses. We deliberately call this an “art”, since pointer arithmetic comes with a lot of pitfalls, but without a safety net. On the other hand, the authors feel that there is also a certain beauty in the minimalism of pointer arithmetic. It’s like driving an oldtimer: it’s loud, it’s difficult to steer, seats are uncomfortable, and there’s no heating. But the heck with it! The oldtimer looks so much better than a modern car. Nevertheless, after driving the oldtimer for a while, it will probably turn out that beauty is not enough, and that safety and usability are more important factors in the long run.

#### Adding integers to pointers

The binary addition operators \(+, -\) are defined for left operands of any pointer type \( T^* \) and right operands of any integral type. Recall that if an array is provided as the left operand, it will implicitly be converted to a pointer using array-to-pointer conversion.

For the behavior of \( + \) to be defined, there must be an array of some length \( n \), such that the left operand \( \text{ptr} \) is a pointer to the element of some index \( k, 0 \leq k \leq n \), in the array. The case \( k = n \) is allowed and corresponds to the situation where \( \text{ptr} \) is a pointer one past the last element of the array (we call this a past-the-end pointer; note that such a pointer must not be dereferenced).

If the second operand \( \text{expr} \) has some value \( i \) such that \( 0 \leq k + i \leq n \), then

\[
\text{ptr} + \text{expr}
\]
is a pointer to the \((k + i)\)-th element of the same array. Informally, we get a pointer that has been moved “i elements to the right” (which actually means to the left if \(i\) is negative). Therefore, if \(p\) is the value of \(\text{ptr}\) (an address), then the value of \(\text{ptr} + \text{expr}\) is the address \(p + s\), assuming that a value of the underlying type occupies \(s\) memory cells. The pleasing fact is that we don’t have to care about \(s\); the operation \(\text{ptr} + \text{expr}\) (which knows \(s\) from the type of \(\text{ptr}\)) does this for us and offers a type-independent way of moving a pointer \(i\) elements to the right.

As before, if \(k + i = n\), we get a past-the-end pointer. Values of \(i\) such that \(k + i\) is not between 0 and \(n\) lead to undefined behavior.

Let us repeat the point that we have made before in connection with random access in Section \[2.6.3\] by default, there are absolutely no checks that the above requirements indeed hold, and it is entirely your responsibility to make sure that this is the case. Failure to do so will result in program crashes, strange behavior of the program, or (probably the worst scenario) seemingly normal behavior, but with the potential of turning into strange behavior at any time, or on any other machine.

Therefore, let us summarize the requirements once more:

- \(\text{ptr}\) must point to the element of index \(k\) in some array of length \(n\), where \(0 \leq k \leq n\), and
- \(\text{expr}\) must have some value \(i\) such that \(0 \leq k + i \leq n\).

Binary subtraction is similar. If \(\text{expr}\) has value \(i\) such that \(0 \leq k - i \leq n\), then

\[
\text{ptr} - \text{expr}
\]

yields a pointer to the array element of index \(k - i\).

The assignment versions \(+=\) and \(-=\) of the two operators can be used with left operands of pointer type as well, with the usual meaning. Similarly, the unary increment and decrement operators \(++\) and \(--\) are available for pointers. Since precedences and associativities are tied to the operator symbols, they are as in Table \[1\] on page \[51\].

Now we can understand the second line of the above code fragment:

```c
bool* const end = crossed_out + n;  // pointer after last element
```

First, the array `crossed_out` is converted to a pointer to its first element (the one of index 0). Since the array has \(n\) elements, adding \(n\) yields a past-the-end pointer `end` for the array. The subsequent loop

```c
for (bool* p = begin; p != end; ++p)
    *p = false; // *p is the element pointed to by p
```

is clear now as well: starting with a pointer \(p\) to the first element (\(p = \text{begin}\)), the element pointed to is set to `false` (*\(p = \text{false}\)*). Then we increment \(p\) so that it points to the next element (\(++p\)). We repeat this as long as \(p\) is different from the past-the-end pointer named `end`. 
CHAPTER 2. FOUNDATIONS

Pointer comparison. We have already discussed the relational operators == and != that simply test whether the two pointers in question point to the same object. But we can also compare two pointers using the operators <, <=, >, and >=. Again, precedences and associativities of all relational operators are as in Table 2 on page 72.

For the result to be specified, there must be an array of some length n, such that the left operand ptr1 is a pointer to the element of some index \( k_1, 0 \leq k_1 \leq n \) in the array, and the second operand ptr2 is a pointer to the element of some index \( k_2, 0 \leq k_2 \leq n \) in the same array. Again, \( k_1 = n \) and \( k_2 = n \) are allowed and correspond to the past-the-end case.

Given this, the result of the pointer comparison is determined by the integer comparison of \( k_1 \) and \( k_2 \). In other words (and quite intuitively), the pointer to the element that comes first in the array is the smaller one.

In our code fragment, the comparison \( p != \text{end} \) could equivalently be replaced by the expression \( p < \text{end} \) which yields \( \text{true} \) as long as \( p \) points to an actual array element, equivalently as long as \( p \) is not a past-the-end pointer.

Comparing two pointers that do not meet the above requirements leads to unspecified results in the four operators <, <=, >, and =>.

Pointer subtraction. There is one more arithmetic operation on pointers. Assume that ptr1 is a pointer to the element of some index \( k_1, 0 \leq k_1 \leq n \) in some array of length n, and the second operand ptr2 is a pointer to the element of some index \( k_2, 0 \leq k_2 \leq n \) in the same array (past-the-end pointers allowed). Then the result of the pointer subtraction

\[
ptr1 - ptr2
\]

is the integer \( k_1 - k_2 \). Thus, pointer subtraction tells us “how far apart” the two array elements are. The behavior of pointer subtraction is undefined if \( ptr1 \) and \( ptr2 \) are not pointers to elements in (or past-the-end pointers of) the same array.

Pointer subtraction (which employs the binary subtraction operator, see Table 1 on page 51 for its specifics) does not occur in the code fragment from the beginning of this section. A typical use is to determine the number of elements in an array that is given by a pointer to its first element and a past-the-end pointer.

Pointer subscripting, or the truth about random access. In reality, the subscript operator [ ] as introduced in Section 2.6.3 does not operate on arrays, but on pointers. Invoking this operator on an array constructs an expression and therefore triggers an array-to-pointer conversion.

Given a pointer \( ptr \) and an expression expr of integral type, the expression

\[
ptr[expr]
\]

is equivalent (also in its requirements on \( ptr \) and \( expr \)) to
2.6. ARRAYS AND POINTERS

*(ptr + expr)

If `expr` has value `i`, the latter expression yields the array element `i` places to the right of the one pointed to by `ptr`. In particular, if `ptr` results from an array-to-pointer conversion, this agrees with the semantics of random access for arrays as introduced in Section 2.6.3.

Table 4 summarizes the new pointer-specific binary operators.

<table>
<thead>
<tr>
<th>Description</th>
<th>Operator</th>
<th>Arity</th>
<th>Prec.</th>
<th>Assoc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>subscript</td>
<td>[]</td>
<td>2</td>
<td>17</td>
<td>left</td>
</tr>
<tr>
<td>dereference</td>
<td>*</td>
<td>1</td>
<td>16</td>
<td>right</td>
</tr>
<tr>
<td>address</td>
<td>&amp;</td>
<td>1</td>
<td>16</td>
<td>right</td>
</tr>
</tbody>
</table>

Table 4: Precedences and associativities of pointer operators. The subscript operator expects rvalues as operands and returns an lvalue. The dereference operator expects an rvalue and returns an lvalue, while the address operator expects an lvalue and returns an rvalue.

What have we gained with pointers? So far it seems that the only use of pointers is to make iteration through an array a little more efficient than iteration by index. But unless we are in the realm of extremely time-critical loops, the savings are marginal. For the sake of readability, we therefore often still use iteration by index. So what is the real justification for the pointer concept?

There are actually two justifications, and one of them will be discussed right away in the next section: pointers are indispensable for getting "practical" arrays with length not known at compile time.

The second justification is not yet around the corner, so we will only briefly touch it here. Arrays are by far not the only containers for sets of data. When we implement data processing algorithms, we should therefore make sure that they work not only for arrays.

For example, finding a container element with a given property (movie theater that plays your desired movie) should be possible for every container that offers the functionality of iterating over its elements. The only uniformity we need is in the iteration process itself.

Every data-processing algorithm of the C++ standard library (we will see some of them later) works in this way: it expects the underlying container to offer iterators conforming to some well-defined iterator concept. The specifics of the container itself are irrelevant for the algorithm.

Here is where pointers come in: they are the iterators offered by arrays. Therefore, even if we don't use pointers in our own code, we have to know about them in order to be able to apply standard library algorithms to arrays.
CHAPTER 2. FOUNDATIONS

2.6.10 Dynamic memory allocation

Let us go back to Program 14 now. Its main drawback is that the number \( n \) is hardwired as 1,000 in this program, just because the length of an array has to be known at compile time.

At least in this respect, arrays are nothing special, though. All types that we have met earlier (\texttt{int}, \texttt{unsigned int}, and \texttt{bool}) have the property that a single object of the type occupies a fixed amount of memory known to the compiler (for example, 32 bits for an \texttt{int} object on many platforms). With arrays, an obvious need arises to circumvent this restriction.

In C++, arrays whose length is determined at runtime can be obtained through \textit{dynamic memory allocation}. Through such an allocation, we create an object with \textit{dynamic} storage duration.

Objects that we have seen so far were all tied to variables, in which case memory gets assigned to them (and is freed again) at predetermined points during program execution (automatic and static storage duration, Section 2.4.3). Objects of dynamic storage duration are not tied to variables, and they may “start to live” (get memory assigned to them) and “die” (get their memory freed) at \textit{any} point during program execution. The programmer can determine these points via \texttt{new} and \texttt{delete} expressions.

The program has some (typically quite large) region of the computer’s main memory available to store dynamically allocated objects. This region is called the \textit{heap}. It is initially unused, but when an object is dynamically allocated, it is being stored on the heap, so that the memory actually used by the program grows.

Here is how this works for Eratosthenes’ Sieve. Remember that we want the list of prime numbers between 2 and \( n - 1 \). The following variant reads the number \( n \) from standard input and dynamically allocates an array of length \( n \). The remainder of the program is as before, except that we explicitly have to free the dynamically allocated storage in the end.

\begin{verbatim}
1 // Program: eratosthenes2.cpp
2 // Calculate prime numbers in \{2,\ldots,n-1\} using
3 // Eratosthenes' sieve.
4
5 #include <iostream>
6
7 int main()
8 {
9    // input
10    std::cout << "Compute prime numbers in \{2,\ldots,n-1\} for n =? ";
11    unsigned int n;
12    std::cin >> n;
13
14    // definition and initialization: provides us with
15    // Booleans crossed_out[0],\ldots, crossed_out[n-1]
\end{verbatim}
bool* const crossed_out = new bool[n]; // dynamic allocation
for (unsigned int i = 0; i < n; ++i)
    crossed_out[i] = false;

// computation and output
std::cout << "Prime numbers in \{2,...,\} and n-1 << ":\n";
for (unsigned int i = 2; i < n; ++i)
    if (!crossed_out[i]) {
        // i is prime
        std::cout << i << " ";
        // cross out all proper multiples of i
        for (unsigned int m = 2*i; m < n; m += i)
            crossed_out[m] = true;
    }
std::cout << "\n";
delete[] crossed_out; // free dynamic memory
return 0;

Program 15: ../progs/lecture/eratosthenes2.cpp

Note that the variable crossed_out is now a pointer rather than an array; after the new declaration, it points to the first element of a dynamically allocated array of length n.

The new expression. For every type T, a new expression can come in any of the following three variants.

- new T
- new T(...)
- new T[expr]

In all cases, the expression returns an rvalue of type T*. Its value is the address of an object of type T that has been dynamically allocated on the heap. The object itself is anonymous, but we usually store the resulting address under a variable name. In Program [15] we call it crossed_out.

In the first and second variant, the effect of the new expression is to dynamically allocate a single object of type T on the heap. Variant 1 leaves the object uninitialized if T is a fundamental type, while variant 2 initializes the new object with whatever appears in parentheses. For example, the following declarations initialize the variables i and j, both of type int*, with the addresses of two new objects of type int.

```
int* i = new int; // *i is undefined
int* j = new int(6); // *j is 6
```
Right now, if we wanted two such objects of type `int`, we’d rather use variables with automatic storage duration and write

```c
int i;    // i is undefined
int j = 6;  // j is 6
```

More interesting for us is the third variant. If `expr` has integer value $n \geq 0$, the effect of the `new` expression is to dynamically allocate an array of length $n$ with underlying type $T$ on the heap. The return value is the address of the first element. This is what we see in line 16 of Program 15.

As usual, the $n$ array elements remain uninitialized if $T$ is a fundamental type.

The `delete` expression. Dynamically allocated memory that is no longer needed should be freed. In C++, the programmer decides at which point this is the case. There are programming languages (Java, for example) that automatically detect and free unused memory on the heap. This automatic process is called garbage collection. It is generally more user-friendly than the manual deletion process in C++, but requires a more sophisticated implementation.

Dynamic storage duration implies that dynamically allocated objects live until the program terminates, unless they are explicitly freed. Dynamically allocated memory is more flexible than static memory, but in return it also involves some administrative effort.

The `delete` expressions take care of freeing memory. They come in two variants.

```c
delete expr
delete[] expr
```

In both variants, `expr` may be a null pointer, in which case the `delete` expression has no effect.

Otherwise, in the first variant, `expr` must be a pointer to a single object that has previously been dynamically allocated with the first or second variant of the `new` expression. The effect is to make the corresponding memory available again for subsequent dynamic allocations on the heap.

For example, at a point in the program where the two `int` objects dynamically allocated through

```c
int* i = new int;  // *i is undefined
int* j = new int(6);  // *j is 6
```

are no longer needed, we would write

```c
delete j;
delete i;
```

The order of deletion does not matter here, but many programmers consider it logical to delete pointers in the inverse order of dynamic allocation: If you need to undo two steps, you first undo the second step.
In the second variant of the delete expression, expr must be a pointer to the first element of an array that has previously been dynamically allocated with the third variant of the new expression. The whole memory occupied by the array is put back on the heap for reuse. This happens in line 32 of Program 15.

If the plain delete is applied to a non-null pointer that does not point to a dynamically allocated single object, the behavior is undefined. The same is true if one tries to delete[] an array where there is only a single object. As always with pointers, the C++ language does not offer any means of detecting such errors.

Memory leaks. Although all memory allocated by a program is automatically freed when the program terminates normally, it is very bad practice to rely on this fact for freeing dynamically allocated memory. If a program does not explicitly free all dynamically allocated memory it is said to have a memory leak. Such leaks are often a sign of bad coding. They usually have no immediate consequences, but without freeing unused storage, a program running for a long time (think of operating system routines) may at some point simply exhaust the available heap storage.

Therefore, we have the following guideline.

**Dynamic Storage Guideline:** new and delete expressions always come in matching pairs.

### 2.6.11 Arrays of characters

Sequences of characters enclosed in double quotes as in

```c++
std::cout << "Prime numbers in \{2,...,\"
```

are called string literals.

So far we have used string literals only within output expressions, but we can work with them in other contexts as well. Most notably, a string literal can be used to initialize an array of characters. Characters are the building blocks of text as we know it. In C++, they are modeled by the fundamental type char that we briefly discuss next.

**The type char.** The fundamental type char represents characters. Characters include the letters a through z (along with their capital versions A through Z), the digits 0 through 9, as well as numerous other special characters like % or $. The line

```c++
char c = 'a';
```

defines a variable c of type char and value 'a', representing the letter a. The expression 'a' is a literal of type char. The quotes around the actual character symbol are necessary in order to distinguish the literal 'a' from the identifier a.

Formally, the type char is an integral type: it has the same operators as the types int or unsigned int, and the C++ standard even postulates a promotion from char to int or unsigned int. It is not specified, though, to which integer the character 'a',
say, will be promoted. Under the widely used ASCII code (American Standard Code for Information Interchange), it is the integer 97.

This setting may not seem very useful, and indeed it makes little sense to divide one character by another. On the other hand, we can for example print the alphabet through one simple loop (assuming ASCII encoding). Execution of the for-loop

```cpp
for (char c = 'a'; c <= 'z'; ++c)
    std::cout << c;
```

writes the character sequence

```
abcdefghijklmnopqrstuvwxyz
```

to standard output. Given this, you may think that the line

```cpp
std::cout << 'a' + 1;
```

prints 'b', but it doesn't. Since the operands of the composite expression 'a' + 1 are of different types, the left operand of type char will automatically be promoted to the more general type int of the right operand. Therefore, the type of the expression 'a' + 1 is int, and its value is 98 (assuming ASCII encoding); and that's what gets printed. If you want 'b' to be printed, you must use the explicit conversion char('a' + 1).

The category of special characters also includes control characters that do something when printed. These are written with a leading backslash, and the most important control character for us is '\n', which causes a line break.

A char value occupies 8 bits (one byte) of memory; whether the value range correspond to the set of integers \{-128, \ldots, 127\} (the signed case) or the set \{0, \ldots, 255\} (the unsigned case) is implementation defined. Since all ASCII characters have integer values in \{0, \ldots, 127\}, they can be represented in both cases.

**From characters to text.** A text is simply a sequence of characters and can be modeled in C++ through an array with underlying type char. For example, the declaration

```cpp
char text[] = {'b', 'o', 'o', 'l'}
```

defines an array of length 4 that represents the text bool.

Alternatively (and more conveniently), we can write

```cpp
char text[] = "bool"
```

This, however, is not equivalent to the former declaration. When an array of characters is initialized with a string literal, the terminating zero character '\0' (of integer value 0) is automatically appended to the array. This character does not correspond to any printable character. After the latter declaration, the array text therefore has length 5. The first four elements are 'b', 'o', 'o', and 'l', and the fifth element is the zero character '\0'.

We call such an array zero-terminated. Unlike normal arrays, zero-terminated arrays "contain" their length. To get this length, we simply have to iterate over the array and count the number of elements before the terminating '\0'.

Here is an application of (arrays of) characters. String matching is the problem of finding the first or all occurrences of a given search string (usually short) in a given text (usually long).

The obvious solution is the following: assuming that the search string has length \( m \), we compare it characterwise with the elements 1, 2, ..., \( m \) of the text. If a mismatch is found for some element, we stop and next compare the search string with the elements 2, 3, ..., \( m + 1 \) of the text, and so on. Sets of \( m \) consecutive elements \( i, i+1, \ldots, i+m-1 \) in the text are called a window.

This algorithm is fast as long as the search string is short, but it may become inefficient for long search strings (see Exercise 85). There is a more sophisticated algorithm (the Knuth-Morris-Pratt algorithm) that is always fast.

The following Program 16 implements the obvious algorithm. It maintains two arrays of characters, one for the search string, and one for the current window. We impose a cyclic order on the window (the first element directly follows the last one); this makes it easy to shift the window one place, by simply replacing element \( i \) of the text with element \( i + m \) (and at the same time advancing the logical first position of the window by one).

```cpp
// Program: string_matching.cpp
// find the first occurrence of a fixed string within the input text, and output the text so far
#include <iostream>
#include <ios> // for std::noskipws
int main () {
    // search string
    const char s[] = "bool";

    // determine search string length m
    unsigned int m = 0;
    for (const char* p = s; *p != '\0'; ++p) ++m;

    // cyclic text window of size m
    char* const t = new char[m];

    // number of characters read so far
    unsigned int w = 0;
    // index where t logically starts
    unsigned int i = 0;

    // find pattern in the text being read from std::cin
    std::cin >> std::noskipws; // don't skip whitespaces!
    for (unsigned int j = 0; j < m;)
```
// compare search string with window at j-th element
if (w < m || s[j] != t[(i+j)%m])
    // input text still too short, or mismatch:
    // advance window by replacing first character
if (std::cin >> t[i]) {
    ++w;
    // one more character read
    j = 0;
    // restart with first characters
    i = (i+1)%m; // of string and window
} else break;
else ++j; // match: go to next character

std::cout << "\n";
delete[] t;
return 0;
}

Program 16: ../../../progs/lecture/string_matching.cpp

When we apply the program to the text of the file eratosthenes.cpp, the program outputs Program 14 up to the first occurrence of the string "bool":

// Program: eratosthenes.cpp
// Calculate prime numbers in \{2,...,n-1\} using
// Eratosthenes' sieve.

#include <iostream>

int main()
{
    const unsigned int n = 1000;

    // definition and initialization: provides us with
    // Booleans crossed_out[0],..., crossed_out[n-1]
    bool

    A few comments need to be made with respect to the handling of standard input here. The program reads the text character by character from std::cin, until this stream becomes "empty". To test this, we use the fact that stream values can implicitly be converted to bool, with the result being true as long as there was no attempt at reading past the end of the stream. Since the value of std::cin >> t[i] is the stream after removal of one character, the conversion to bool exactly tells us whether there still was a character in the stream, or not.

    Most conveniently, the program is run by redirecting standard input to a file containing the text. In this case, the stream std::cin will become empty exactly at the end of the file. The line
std::cin >> std::noskipws; // don't skip whitespaces!
is necessary to tell the stream that whitespaces (blanks, newlines, etc.) should not be ignored (by default, they are). This allows us to search for strings that contain whitespaces, and it allows us to output the text (up to the first occurrence of the search string) in its original layout.

### 2.6.12 Multidimensional arrays

In C++, we can have arrays of arrays. For example, the declaration

```c++
int a[2][3]
```

declares a to be an array of length 2 whose elements are arrays of length 3 with underlying type int. We also say that a is a *multidimensional array* (in this case of dimensions 2 and 3). The type of a is “int[2][3]”, and the underlying type is int[3]. In general, the declaration

\[ T \text{a[expr1]...[exprk]} \]

defines an array a of length \( n_1 \) (value of expr1) whose elements are arrays of length \( n_2 \) (value of expr2) whose elements are...you get the picture. The values \( n_1, \ldots, n_k \) are called the *dimensions* of the array, and the expressions \( \text{expr1}, \ldots, \text{exprk} \) must be constant expressions of integral type and positive value.

Random access in multidimensional arrays works as expected: \( \text{a[i]} \) is the element of index i, and this element is an array itself. Consequently, \( \text{a[i][j]} \) is the element of index j in the array \( \text{a[i]} \), and so on.

Although we usually think of multidimensional arrays as tables or matrices, the memory layout is “flat” like for one-dimensional arrays. For example, the twodimensional array declared through `int a[2][3]` occupies a contiguous part of the memory, with space for \( 6 = 2 \times 3 \) objects of type int, see Figure [11](Figure 11: Memory layout of a twodimensional array)

```
<table>
<thead>
<tr>
<th>a[0][0]</th>
<th>a[0][1]</th>
<th>a[0][2]</th>
<th>a[1][0]</th>
<th>a[1][1]</th>
<th>a[1][2]</th>
</tr>
</thead>
</table>

a[0]        a[1]        
```

Multidimensional arrays can be initialized in a way similar to onedimensional arrays; the value for the *first* (and *only* the first) dimension may be omitted:

```c++
int a[][3] = { {2,4,6}, {1,3,5} };
```

This defines an array of type int[2][3] where \( \{2,4,6\} \) is used to initialize the element \( \text{a[0]} \), and \( \{1,3,5\} \) is used for \( \text{a[1]} \).
Dynamic allocation of multidimensional arrays. The required dimensions of a multidimensional array may not be known at compile time in which case dynamic allocation is called for. Let us start with the case where all dimensions but the first are known at compile time. If \( \text{expr} \) has value \( n \geq 0 \), a pointer to a dynamically allocated array of length \( n \) with underlying type \( T[n_2]...[n_k] \) is obtained from a new expression

\[
\text{new } T[\text{expr}][\text{expr2}][\text{expr}...][\text{exprk}]
\]

where \( \text{expri} \) has value \( n_i, \ i = 1, \ldots, k \). All dimensions but the first must be constant expressions. If you think about it for a minute, this is not surprising. For example, in order to generate machine language code for random access operations on the dynamically allocated array, the compiler must know how many memory cells a single element of the underlying type \( T[n_2]...[n_k] \) occupies (see Section [2.6.3]). But this is only possible if the values \( n_2, \ldots, n_k \) are known at compile time.

Pointers to arrays. If we want to use the above new expression to initialize a pointer variable (with the address of the first element of the multidimensional array), we need the type “pointer to \( T[n_2]...[n_k] \)” . As you may suspect, we informally call this type “\( T[n_2]...[n_k]* \)”, but we can’t write it like that in C++, since \( T[n_2]...[n_k] \) is not a type name. Again, we have to resort to an implicit definition of the desired pointer variable \( p \), as in the following code fragment.

```c
int n = 2;
```

The parentheses are necessary here, since int *p[3] (which is the same as int* p[3]) declares \( p \) to be an array of pointers to int (see also next paragraph). C++ syntax is bittersweet.

Arrays of pointers. If you’re asking for a multidimensional array with non-constant dimensions among \( n_2, \ldots, n_k \), the official answer is: there is none. But under the counter, you can buy a very good imitation.

One first solution that suggests itself when you reconsider the flat memory layout of multidimensional arrays is this: you dynamically allocate a onedimensional array of length \( n = n_1 \times n_2 \times \cdots \times n_k \) and artificially partition it into subarrays by doing some juggling with indices.

Let us discuss the twodimensional case only to avoid lengthy formulae. A twodimensional array with dimensions \( n \) and \( m \) can be simulated by a onedimensional array of length \( n \times m \). The element with logical indices \( i \in [0, 1, \ldots, n-1] \) and \( j \in [0, 1, \ldots, m-1] \) appears at index \( mi + j \) in the onedimensional array. Vice versa, the element of index \( \ell \) in the onedimensional array has logical indices \( i = \ell \div m \) and \( j = \ell \mod m \). This works because the function

\[
(i, j) \rightarrow mi + j
\]
bijectively maps the set of logical indices \((i, j)\) to the set of numbers \(\{0, 1, \ldots, nm - 1\}\). Intuitively, this mapping flattens the imaginary table of \(n\) rows and \(m\) columns by simply putting one row after another. As you can see from Figure 11, this is exactly what the compiler is implicitly doing for multidimensional arrays with constant dimensions \(n_1, \ldots, n_k\).

Doing it explicitly for non-constant dimensions is only a workaround, though, since we lose the intuitive notation \(a[i][j]\); moreover, this workaround becomes even more cumbersome with higher dimensional arrays.

A better solution that keeps the notation \(a[i][j]\) and that smoothly extends to higher dimensions is the following (again, we only discuss the case of a twodimensional array with dimensions \(n\) and \(m\)): you first dynamically allocate one array of \(n\) pointers, and then you let every single pointer point to the first element of an individual, dynamically allocated array of length \(m\). The following code fragment demonstrates this.

```c
// a points to the first element of an array of n pointers to int
int** a = new int*[n];
for (int i = 0; i < n; ++i)
    // a[i] points to the first element of an array of m int's
    a[i] = new int[m];
```

The type \(\text{int*}\) is “pointer to pointer to int”. \(a[i]\) is therefore a pointer to int (see the paragraph on pointer subscripting in Section 2.6.9), and \(a[i][j]\) is an lvalue of type int, just like in a “regular” twodimensional array.

The memory layout is different, though: Figure 11 is replaced by Figure 12. This means, the twodimensional array is patched up from a set of \(n\) onedimensional arrays, but these \(n\) arrays are not necessarily consecutively arranged in memory. In fact, the \(n\) arrays may even have different lengths. This is useful for example when you want to store a lower-triangular matrix; in this case, it suffices if the row of index \(i\) has length \(i + 1\).

```
\begin{verbatim}
  a[0]  a[1]  a[1][0]  a[1][1]  a[1][2]  a[0][0]  a[0][1]  a[0][2]
\end{verbatim}
```

Figure 12: Memory layout of a twodimensional array realized by an array of pointers

Computing shortest paths. Let us conclude this section with an interesting application of (multidimensional) arrays. Imagine a rectangular factory floor, subdivided into square cells. Some of the cells are blocked with obstacles (these could for example be machines
or cupboards, but let us abstractly call them “walls”). A robot is initially located at some cell $S$ (the source), and the goal is to move the robot to some other cell $T$ (the target). At any time, the robot can make one step from its current cell to any of the four adjacent cells, but for obvious reasons it may only use cells that are empty.

Given this setup, we want to find a shortest possible robot path from $S$ to $T$ (or find out that no such path exists). Here, the length of a robot path is the number of steps taken by the robot during its motion from $S$ to $T$ (the initial cell $S$ does not count; in particular, it takes 0 steps to reach $S$ from $S$). Figure 13 (left) shows an example with $8 \times 12$ cells.

![Figure 13: Left: What is a shortest robot path from $S$ to $T$? Right: This one!](image)

In this example, a little thinking reveals that there are essentially two different possibilities for the robot to reach $T$: it can pass below the component of walls adjacent to $S$, or above. It turns out that passing above is faster, and a resulting shortest path (of length 21) is depicted in Figure 13 (right). Note that in general there is not a unique shortest path. In our example, the final right turn of the path could also have been made one or two cells further down.

We want to write a program that finds a shortest robot path, given the dimensions $n$ (number of rows) and $m$ (number of columns) of the factory floor, the coordinates of source and target, and the walls. How can this be done? Before reading further, we encourage you to think about this problem for a while. Please note that the brute-force approach of trying all possible paths and selecting the shortest one is not an option, since the number of such paths is simply too large already for moderate floor dimensions. (Besides, how do you even generate all these paths?)

Here is an approach based on dynamic programming. This general technique is applicable to problems whose solutions can quickly be obtained from the solutions to smaller subproblems of the same structure. The art in dynamic programming is to find the “right” subproblems, and this may require a more or less far-reaching generalization of the original problem.

Once we have identified suitable subproblems, we solve all of them in turn, from the smaller to the larger ones, and memorize the solutions. That way, we have all the
information that we need in order to quickly compute the solution to a given subproblem from the solutions of the (already solved) smaller subproblems.

In our case, we generalize the problem as follows: for all empty cells \( C \) on the floor, compute the length of a shortest path from \( S \) to \( C \) (where the value is \( \infty \) if no such path exists). We claim that this also solves our original problem of computing a shortest path from \( S \) to \( T \): Assume that the length of a shortest path from \( S \) to \( T \) is \( \ell < \infty \) (otherwise we know right away that there is no path at all). We also say that \( T \) is reachable from \( S \) in \( \ell \) steps.

Now if \( T \neq S \), there must be a cell adjacent to \( T \) that is reachable from \( S \) in \( \ell - 1 \) steps, and adjacent to this a cell reachable in \( \ell - 2 \) steps etc. Following such a chain of cells until we get to \( S \) gives us a path of length \( \ell \) which is shortest possible.

Let us rephrase the generalized problem: we want to label every empty cell \( C \) with a nonnegative integer (possibly \( \infty \)) that indicates the length of a shortest path from \( S \) to \( C \). Here are the subproblems to which we plan to reduce this: for a given integer \( i \geq 0 \), label all the cells that are reachable from \( S \) in at most \( i \) steps. For \( i = nm - 1 \) (actually, for some smaller value), this labels all cells that are reachable from \( S \) at all, since a shortest path will never enter a cell twice.

Here is the reduction from larger to smaller subproblems: assume that we have already solved the subproblem for \( i - 1 \), i.e. we have labeled all cells that are reachable from \( S \) within \( i - 1 \) or less steps. In order to solve the subproblem for \( i \), we still need to label the cells that are reachable in \( i \) steps (but not less). But this is simple, since these cells are exactly the unlabeled ones adjacent to cells with label \( i - 1 \).

Figure 14 illustrates how the frontier of labeled cells grows in this process, for \( i = 0, 1, 2, 3 \).

Continuing in this fashion, we finally arrive at the situation depicted in Figure 15: all empty cells have been labeled (and are in fact reachable from \( S \) in this example). To find a shortest path from \( S \) to \( T \), we start from \( T \) (which has label 21) and follow any path of decreasing labels (20, 19, \ldots) until we finally reach \( S \).

The shortest path program. Let’s get to the C++ implementation of the above method. We represent the floor by a dynamically allocated twodimensional array \( \text{floor} \) with dimensions \( n + 2 \) and \( m + 2 \) and entries of type int. (Formally, \( \text{floor} \) is a pointer to the first element of an array of \( n + 2 \) pointers to int, but we still call this a twodimensional array). These dimensions leave space for extra walls surrounding the floor. Such extra walls allow us to get rid of special cases: floor cells having less than four adjacent cells. In general, an artificial data item that guards the actual data against special cases is called a sentinel.

The heart of the program (which appears as Program 17 below) is a loop that computes the solution to subproblem \( i \) from the solution to subproblem \( i - 1 \), for \( i = 1, 2, \ldots \). The solution to subproblem 0 is readily available: we set the \( \text{floor} \) entry corresponding to \( S \) to 0, and the entries corresponding to the empty cells to -1 (this is meant to indicate that the cell has not been labeled yet). Walls are always labeled with the integer -2.
The solution to subproblem i labels all cells C reachable from S within at most i steps with the length of the shortest path from S to C.

In iteration i of the loop, we simply go through all the yet unlabeled cells and label exactly the ones with i that have an adjacent cell with label i − 1. The loop terminates as soon as no progress is made anymore, meaning that no new cell could be labeled in the current iteration. Here is the code.

```c
// main loop: find and label cells reachable in i=1,2,... steps
for (int i=1; ++i) {
    bool progress = false;
    for (int r=1; r<n+1; ++r)
        for (int c=1; c<m+1; ++c) {
            if (floor[r][c] != -1) continue; // wall, or labeled before
            // is any neighbor reachable in i-1 steps?
            if (floor[r-1][c] == i-1 || floor[r+1][c] == i-1 ||
                floor[r][c-1] == i-1 || floor[r][c+1] == i-1 ) {
                floor[r][c] = i; // label cell with i
                progress = true;
            }
        }
}
```
Figure 15: The solution to subproblem $i = 23$ solves the generalized problem and the original problem (a shortest path is obtained by starting from $T$ and following a path of decreasing labels).

...
CHAPTER 2. FOUNDATIONS

8 12
------X------
-XXX--X------
--SX---------
--X--XXX--
--X--X----
--X--X----
--X--X-T--
-------X----

Figure 16: Input for Program 17 corresponding to the example of Figure 18

// assign initial floor values from input:
// source: 'S' -> 0 (source reached in 0 steps)
// target: 'T' -> -1 (number of steps still unknown)
// wall: 'X' -> -2
// empty cell: '-' -> -1 (number of steps still unknown)
for (int r =1; r<n+1; ++r)
    for (int c =1; c<m+1; ++c) {
        char entry = '-';
        std::cin >> entry;
        if (entry == 'S') floor[r][c] = 0;
        else if (entry == 'T') floor[tr = r][tc = c] = -1;
        else if (entry == 'X') floor[r][c] = -2;
        else if (entry == '-') floor[r][c] = -1;
    }

Now we add the surrounding walls as sentinels.

// add surrounding walls
for (int r =0; r<n+2; ++r)
    floor[r][0] = floor[r][m+1] = -2;
for (int c =0; c<m+2; ++c)
    floor[0][c] = floor[n+1][c] = -2;

Next comes the main loop that we have already discussed above. It labels all reachable
cells, so that we obtain a labeling as in Figure 15. From this labeling, we must now extract
the shortest path from S to T. As explained above, this can be done by following a chain
of adjacent cells with decreasing labels. For every cell on this path (except S), we put
the integer −3 into the corresponding floor entry; this allows us to draw the path in
the subsequent output. If no path was found (or if there is no target), the body of the
while statement in the following code fragment is (correctly) not executed at all.

// mark shortest path from source to target (if there is one)
int r = tr; int c = tc; // start from target
while (floor[r][c] > 0) {
    const int d = floor[r][c] - 1; // distance one less
    floor[r][c] = -3; // mark cell as being on shortest path
    // go to some neighbor with distance d
    if (floor[r-1][c] == d) --r;
    else if (floor[r+1][c] == d) ++r;
    else if (floor[r][c-1] == d) --c;
    else ++c; // (floor[r][c+1] == d)
}

Finally, the output: we map the integer entries of floor back to characters, where
−3 becomes 'o', our path symbol. Inserting \n at the right places, we obtain a copy
of the input floor, with the shortest path appearing in addition. We must also not forget
to delete the dynamically allocated arrays in the end.

// print floor with shortest path
for (int r=1; r<n+1; ++r) {
    for (int c=1; c<m+1; ++c)
        if (floor[r][c] == 0) std::cout << 'S';
        else if (r == tr & c == tc) std::cout << 'T';
        else if (floor[r][c] == -3) std::cout << 'o';
        else if (floor[r][c] == -2) std::cout << 'X';
        else std::cout << '-';
    std::cout << "\n";
}
// delete dynamically allocated arrays
for (int r=0; r<n+2; ++r)
    delete[] floor[r];
delete[] floor;

return 0;

In case of our initial example, the output looks like in Figure 17. Program 17 shows
the complete source code.

1 #include <iostream>
2
3 int main()
4 {
5     // read floor dimensions
6     int n; std::cin >> n; // number of rows
7     int m; std::cin >> m; // number of columns
8
9     // dynamically allocate twodimensional array of dimensions
10     // (n+2) x (m+2) to hold the floor plus extra walls around
int** const floor = new int*[n+2];
for (int r=0; r<n+2; ++r)
    floor[r] = new int[m+2];

// target coordinates, set upon reading 'T'
int tr = 0;
int tc = 0;

// assign initial floor values from input:
// source: 'S' -> 0 (source reached in 0 steps)
// target: 'T' -> -1 (number of steps still unknown)
// wall: 'X' -> -2
// empty cell: '-' -> -1 (number of steps still unknown)
for (int r=1; r<n+1; ++r)
    for (int c=1; c<m+1; ++c) {
        char entry = '-';
        std::cin >> entry;
        if (entry == 'S') floor[r][c] = 0;
        else if (entry == 'T') floor[tr = r][tc = c] = -1;
        else if (entry == 'X') floor[r][c] = -2;
        else if (entry == '-') floor[r][c] = -1;
    }

// add surrounding walls
for (int r=0; r<n+2; ++r)
    floor[r][0] = floor[r][m+1] = -2;
for (int c=0; c<m+2; ++c)
    floor[0][c] = floor[n+1][c] = -2;

// main loop: find and label cells reachable in i=1,2,... steps
for (int i=1; ++i) {
    bool progress = false;
for (int r=1; r<n+1; ++r)
    for (int c=1; c<m+1; ++c) {
        if (floor[r][c] != -1) continue; // wall, or labeled before
        // is any neighbor reachable in i-1 steps?
        if (floor[r-1][c] == i-1 || floor[r+1][c] == i-1 ||
            floor[r][c-1] == i-1 || floor[r][c+1] == i-1 ) {
            floor[r][c] = i; // label cell with i
            progress = true;
        }
    }
    if (!progress) break;
}

// mark shortest path from source to target (if there is one)
int r = tr; int c = tc; // start from target
while (floor[r][c] > 0) {
    const int d = floor[r][c] - 1; // distance one less
    floor[r][c] = -3; // mark cell as being on shortest path
    // go to some neighbor with distance d
    if (floor[r-1][c] == d) --r;
    else if (floor[r+1][c] == d) ++r;
    else if (floor[r][c-1] == d) --c;
    else ++c; // (floor[r][c+1] == d)
}

// print floor with shortest path
for (int r=1; r<n+1; ++r) {
    for (int c=1; c<m+1; ++c)
        if (floor[r][c] == 0) std::cout << 'S';
        else if (r == tr && c == tc) std::cout << 'T';
        else if (floor[r][c] == -3) std::cout << 'o';
        else if (floor[r][c] == -2) std::cout << 'X';
        else std::cout << '-';
    std::cout << "\n";
}

// delete dynamically allocated arrays
for (int r=0; r<n+2; ++r)
    delete[] floor[r];
delete[] floor;
return 0;

Program 17: ../progs/lecture/shortest_path.cpp
2.6.13 Beyond arrays and pointers

Arrays are very useful for many tasks and allow us to solve nontrivial problems like finding shortest paths in the previous section. From a theoretical point of view, arrays are in fact the only containers that we need.

On the other hand, there are two main drawbacks of arrays that we want to recapitulate here.

**Arrays have fixed length.** Any array, even if it is dynamically allocated, has a fixed length. In other words, we have to know before defining or dynamically allocating an array how many elements we need to store in it. Often, this is unrealistic. For example, in some application we might need to store a sequence of input numbers, but we don’t know in advance how many numbers we will get. A typical “solution” is to dynamically allocate a very large array and just hope that the sequence fits in. The problems with this and a better (but still cumbersome) solution are outlined in Exercise 88.

A “real” solution is possible in C++ through the use of *vectors*. These are containers from the standard library that combine the classical array functionality (and its efficiency) with the possibility of growing (and shrinking) in length. Vectors can be implemented on top of arrays, and they have something similar to the mechanism outlined in Exercise 88 “built in”. Vectors also largely remove the necessity of working with pointers. We will get to vectors (and their realization) later in this book.

**Arrays are insecure.** The usage of out-of-bound array indices is not detected in C++, and the same holds for pointers to addresses where no object lives. With some care, you can write small programs that use arrays and pointers in a correct manner, but in complex programs, this is not easy at all. Debugging facilities of modern compilers can help, but even well-tested and frequently used large programs do not necessarily get it right. In fact, some people (let’s call them attackers) are making a business of exploiting programming errors related to arrays and pointers in order to create malicious software.

Suppose that the attacker knows that some program—think of an operating system routine or a webserver—may (unintentionally) write input data beyond the bounds of an array. Due to the von-Neumann architecture, the part of the main memory being accidentally modified in this way may contain the actual program instructions. The attacker may then be able to prepare an input to the program in such a way that the program modifies itself to do whatever the attacker wants it to do. This modification runs with the same access rights as the original one, and these might be administrator rights in the worst case.

In this way, an attacker could “hijack” the computer that runs the program, and subsequently misuse it for illegal activities like sending spam, or paralyzing web servers by flooding them with requests.

For us that we are not (yet) professional programmers, the security aspect is less of a concern here. More important is that programming errors due to improper use of arrays and pointers can be very hard to find and often remain undetected until they suddenly...
result in strange and seemingly inexplicable behavior of the program. Also here, using vectors instead of arrays helps, since there are many potential errors related to arrays and pointers that you simply cannot make with vectors.

**Why arrays, after all?** Now you may ask why we have introduced arrays and pointers at all when there are more flexible and safer alternatives. Here are the three reasons.

1. Arrays and pointers are the simplest models of important standard library concepts (container and iterator).

2. Unlike vectors, arrays can be introduced without the need to discuss syntactical and semantical aspects of C++ functions and classes (that we simply don’t have at our disposal at this point);

3. In order to really understand later how standard library containers and iterators are realized, it is necessary to know about arrays and pointers.

The take-home message here is this: it is important to get familiar with the concepts behind arrays and pointers, but it is less important to be able to actually program with arrays and pointers on a large scale.

### 2.6.14 Details

**Command line arguments.** In Program 16 for string matching, it is not very convenient that the search string is fixed. We then have to recompile the program every time we want to search for another string.

A more flexible alternative is to pass the search string as a command line argument that we provide upon calling the program.

The main function can access such command line arguments if we provide suitable parameters. Here is how the first ten lines of Program 16 have to be changed in order to make this work.

```cpp
1 // Program: string_matching2.cpp
2 // find the first occurrence of a string (provided as command
3 // line argument) within the input text, and output text so far
4
5 #include<iostream>
6 #include<ios> // for std::noskipws
7
8 int main (int argc, char* argv[])
9 {
10   if (argc < 2) {
11     // no command line arguments (except program name)
12     std::cout << "Usage: string_matching2 <string>\n";
13     return 1;
```
14  }
15
16  // search string: second command line argument
17  const char* const s = argv[1];

    The values of argc and and argv[] (which is an array of pointers each of which in turn points to the first elements of a zero-terminated array of characters) are initialized by the operating system when it calls the main function. We will explain function parameters in detail later, here we will be satisfied with an example. Suppose that we call the program like this (assuming a Unix-type system):

    ./string_matching2 bool

    Then argc (which counts the number of command line arguments) gets initialized with value 2. This count includes the program name itself ("string_matching2" in this case), and any additional strings provided on the command line (just the single string "bool" in this case). The 2 arrays argv[0] and argv[1] get initialized with the strings "string_matching2" and "bool" as described in Section 2.6.11 above. Consequently, after its definition, the pointer variable s in the above piece of code points to the first element of a zero-terminated array of characters that corresponds to the string "bool". This gets us back to the situation in Program 16 after line 10, and the remainders of both programs are identical.

2.6.15 Goals

Dispositional. At this point, you should ...

1) know what an array is, and what random access and iteration mean in the context of arrays;
2) understand the pointer concept, and how to compute with addresses;
3) be aware that (and understand why) arrays and pointers must be used with care;
4) know that characters and arrays of characters can be used to perform basic text processing tasks;
5) know that (multidimensional) arrays of variable length can be obtained by dynamic memory allocation;

Operational. In particular, you should be able to ...

(G1) read, understand, and argue about simple programs involving arrays and pointers; relevant aspects are in particular pointer arithmetic and const-correctness

(G2) write programs that define array variables or dynamically allocate (multidimensional) arrays;

(G3) write programs that read a sequence of data into a (dynamically allocated / multidimensional) array;
2.6. ARRAYS AND POINTERS

(G4) write programs that perform simple data processing tasks by using random access in (multidimensional) arrays as the major tool;

(G5) within programs, iterate over a (dynamically allocated / multidimensional) array by using pointer arithmetic;

(G6) write programs that perform simple text processing tasks with arrays of characters;

2.6.16 Exercises

Exercise 77

a) What does the following program output, and why?

```cpp
#include <iostream>

int main()
{
    int a[] = {5, 6, 2, 3, 1, 4, 0};
    int* p = a;
    do {
        std::cout << *p << " ";
        p = a + *p;
    } while (p != a);

    return 0;
}
```

b) More generally, suppose that in the previous program, a is initialized with some sequence of n different numbers in \( \{0, \ldots, n-1\} \) (we see this for \( n = 7 \) in the previous program). Prove that the program terminates in this case.

Exercise 78 Assume that in some program, a is an array of underlying type int and length n.

a) Given a variable i of type int with value \( 0 \leq i \leq n \), how can you obtain a pointer p to the element of index i in a? (Note: if \( i = n \), this is asking for a past-the-end pointer.)

b) Given a pointer p to some element in a, how can you obtain the index i of this element? (Note: if p is a past-the-end pointer, the index is defined as n.)

Write code fragments that compute p from i in a) and i from p in b).
Exercise 79 Find and fix at least 5 problems in the following program. The fixed program should indeed correctly do what it claims to do. Is the fixed program const-correct? If not, make it const-correct! (This is a theory exercise, but you may of course use the computer to help you.)

```
#include <iostream>

int main()
{
    int a[7] = {0, 6, 5, 3, 2, 4, 1}; // static array
    int* const b = new int[7];

    // copy a into b using pointers
    for (int* p = a; p <= a + 7; ++p)
        *b++ = *p;

    // cross-check with random access
    for (int i = 0; i <= 7; ++i)
        if (a[i] != b[i])
            std::cout << "Oops, copy error...\n";

    delete b;

    return 0;
}
```

Exercise 80 Let us call a natural number $k$-composite if and only if it is divisible by exactly $k$ different prime numbers. For example, prime powers are 1-composite, and $6 = 2 \cdot 3$ as well as $20 = 2 \cdot 2 \cdot 5$ are 2-composite. Write a program `k_composite.cpp` that reads numbers $n \geq 0$ and $k \geq 0$ from the input and then outputs all $k$-composite numbers in $\{2, \ldots, n - 1\}$. How many 7-composite numbers are there for $n = 1,000,000$? (G2)(G4)

Exercise 81 Write a program `inverse_matrix.cpp` that inverts a $3 \times 3$ matrix $A$ with real entries. The program should read the nine matrix entries from the input, and then output the inverse matrix $A^{-1}$ (or the information that the matrix $A$ is not invertible). In addition, the program should output the matrix $AA^{-1}$ in order to let the user check whether the computation of the inverse was accurate (in the fully accurate case, the latter product is the identity matrix).

**Hint:** For the computation of the inverse, you can employ Cramer's rule. Applied to the computation of the inverse, it yields that $A^{-1}_{ij}$ (the entry of $A^{-1}$ in row $i$ and column $j$) is given by

$$A^{-1}_{ij} = \frac{(-1)^{i+j}\det(A^i)}{\det(A)},$$
where \( \det(M) \) is the determinant of a square matrix \( M \), and \( A_{ji} \) is the \( 2 \times 2 \) matrix obtained from \( A \) by deleting row \( j \) and column \( i \).

To compute the determinant of a \( 3 \times 3 \) matrix, you might want to use the well-known Sarrus’ rule.

**Exercise 82** Write a program read_array that reads a sequence of \( n \) integers from standard input into an array. The number \( n \) is the first input, and then the program expects you to input another \( n \) values. After reading the \( n \) values, the program should output them in the same order. (If you can do this, you have proven that you are no longer a complete novice, according to Stroustrup.) For example, on input \( 5 \ 4 \ 3 \ 6 \ 1 \ 2 \) the program should output \( 4 \ 3 \ 6 \ 1 \ 2 \).

**Exercise 83** Enhance the program read_array.cpp from Exercise 82 so that the resulting program sort_array.cpp sorts the array elements into ascending order before outputting them. Your sorting algorithm does not have to be particularly efficient, the main thing here is that it works correctly. Test your program on some larger inputs (preferably read from a file, after redirecting standard input). For example, on input \( 5 \ 4 \ 3 \ 6 \ 1 \ 2 \) the program should output \( 1 \ 2 \ 3 \ 4 \ 6 \).

**Exercise 84** Enhance the program read_array.cpp from Exercise 82 so that the resulting program cycles.cpp interprets the input sequence of \( n \) integers as a permutation \( \pi \) of \( \{0, \ldots, n-1\} \), and that it outputs the cycle decomposition of \( \pi \).

Some explanations are in order: a permutation \( \pi \) is a bijective mapping from the set \( \{0, \ldots, n-1\} \) to itself; therefore, the input sequence can be interpreted as the sequence of values \( \pi(0), \ldots, \pi(n-1) \) of a permutation \( \pi \) if and only if it contains every number from \( \{0, \ldots, n-1\} \) exactly once.

The program cycles.cpp should first check whether the input sequence satisfies this condition, and if not, terminate with a corresponding message. If the input indeed encodes a permutation \( \pi \), the program should output the cycle decomposition of \( \pi \). A cycle in \( \pi \) is any sequence of the form \( (n_1 \ n_2 \ \cdots \ n_k) \) such that

- \( n_2 = \pi(n_1), \ n_3 = \pi(n_2), \ \ldots, \ n_k = \pi(n_{k-1}), \) and \( n_1 = \pi(n_k), \) and
- \( n_1 \) is the smallest element among \( n_1, \ldots, n_k. \)

Any cycle uniquely determines the \( \pi \)-values of all its elements; on the other hand, every element appears in some cycle (which might be of the trivial form \( (n_1) \), meaning that \( \pi(n_1) = n_1) \). This implies that the permutation decomposes into a unique set of cycles. For example, the permutation \( \pi \) given by

\[
\pi(0) = 4, \ \pi(1) = 2, \ \pi(2) = 3, \ \pi(3) = 1, \ \pi(4) = 0
\]

decomposes into the two cycles \((0 \ 4)\) and \((1 \ 2 \ 3)\).
**Exercise 85** Consider the string matching algorithm of Program 16. Prove that for all $m > 1$, $n \geq m$, there exists a search string $s$ of length $m$ and a text $t$ of length $n$ on which the algorithm in Program 16 performs $m(n - m + 1)$ comparisons between single characters.  

**Exercise 86** Consider the following program that defines and initializes a threedimensional array.

```cpp
#include <iostream>

int main()
{
    int a[4][2][3] =
    {
        { // the 4 elements of a:
            { // the 2 elements of a[0]:
                {2, 4, 5}, // the three elements of a[0][0]
                {4, 6, 7} // the three elements of a[0][1]
            },
            { // the 2 elements of a[1]:
                {1, 5, 9}, // the three elements of a[1][0]
                {4, 6, 1} // the three elements of a[1][1]
            },
            { // the 2 elements of a[2]:
                {5, 9, 0}, // the three elements of a[2][0]
                {1, 5, 3} // the three elements of a[2][1]
            },
            { // the 2 elements of a[3]:
                {6, 7, 7}, // the three elements of a[3][0]
                {7, 8, 5} // the three elements of a[3][1]
            }
    };

    return 0;
}
```

Write a program threedim_array.cpp that enhances this program by a (nested) loop that iterates over the array $a$ and its subarrays to output all the 24 int values that are stored in $a$ and its subarrays. Do not use random access to do this but pointer arithmetic.  

**Exercise 87** Write a program frequencies.cpp that reads a text from standard input (like in Program 16) and outputs the frequencies of the letters in the text, where we do not distinguish between lower and upper case letters. For this exercise, you may assume that the type `char` implements ASCII encoding. This means that all characters have integer values in $\{0, 1, \ldots, 127\}$. Moreover, in ASCII, the values of
the 26 upper case literals 'A' up to 'Z' are consecutive numbers in \{65, \ldots , 90\}; for 
the lower case literals 'a' up to 'z', the value range is \{97, \ldots , 122\}.

Running this on the lyrics of *Yesterday* (The Beatles) for example should yield the 
following output.

<table>
<thead>
<tr>
<th>Frequencies:</th>
<th>i: 27 of 520</th>
<th>r: 19 of 520</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: 45 of 520</td>
<td>j: 0 of 520</td>
<td>s: 36 of 520</td>
</tr>
<tr>
<td>b: 5 of 520</td>
<td>k: 3 of 520</td>
<td>t: 31 of 520</td>
</tr>
<tr>
<td>c: 5 of 520</td>
<td>l: 20 of 520</td>
<td>u: 9 of 520</td>
</tr>
<tr>
<td>d: 28 of 520</td>
<td>m: 10 of 520</td>
<td>v: 6 of 520</td>
</tr>
<tr>
<td>e: 65 of 520</td>
<td>n: 30 of 520</td>
<td>w: 19 of 520</td>
</tr>
<tr>
<td>f: 4 of 520</td>
<td>o: 43 of 520</td>
<td>x: 0 of 520</td>
</tr>
<tr>
<td>g: 13 of 520</td>
<td>p: 4 of 520</td>
<td>y: 34 of 520</td>
</tr>
<tr>
<td>h: 27 of 520</td>
<td>q: 0 of 520</td>
<td>z: 0 of 520</td>
</tr>
<tr>
<td></td>
<td>Other: 37 of 520</td>
<td></td>
</tr>
</tbody>
</table>

2.6.17 Challenges

Exercise 88 The fact that an array has a fixed length is often inconvenient. For example, in Exercise 82 and in Exercise 83, the number of elements to be read into the array had to be provided as the first input in order for the program to be able to dynamically allocate an array of the appropriate length. But in practice, the length of the input sequence is often not known a priori.

We would therefore like to write a program that reads a sequence of integers from standard input into an array, where the length of the sequence is not known beforehand (and not part of the input)—the program should simply read one number after another until the stream becomes empty.

One possible strategy is to dynamically allocate an array of large length, big enough to store any possible input sequence. But if the sequence is short, this is a huge waste of memory, and if the sequence is very long, the array might still not be large enough.

a) Write a program read_array2.cpp that reads a sequence of integers of unknown length into an array, and then outputs the sequence. The program should satisfy the following two properties.

(i) The amount of dynamically allocated memory in use by the program should at any time be proportional to the number of sequence elements that have been read so far. To be concrete: there must be a positive constant \(a\) such that no more than \(ak\) cells of dynamically allocated memory are in use when \(k\) elements have been read, \(k \geq 1\). We refer to this property as space efficiency. It ensures that even very long sequences can be read (up to the applicable memory limits), but that short sequences consume only little memory.
(ii) The number of assignments (of values to array elements) performed so far should at any time be proportional to the number of sequence elements that have been read so far, with the same meaning of proportionality as above. We refer to this property as time efficiency. It ensures that the program is only by a constant factor slower than the program read_array.cpp that knows the sequence length in advance.

b) Determine the constants of proportionality $a$ for properties (i) and (ii) of your program.

Exercise 89 For larger floors, Program [17] can become quite inefficient, since every step $i$ examines all cells of the floor in order to find the (possibly very few) ones that have to be labeled with $i$ in that step. A better solution would be to examine only the neighbors of the cells that are already labeled with $i-1$, since only these are candidates for getting label $i$.

Write a program shortest_path_fast.cpp that realizes this idea, and measure the performance gain on some larger floors of your choice.

Exercise 90 In 1772, Leonhard Euler discovered the quadratic polynomial

$$n^2 + n + 41$$

with the following remarkable property: if you evaluate it for $n = 0, 1, \ldots, 39$, you always get a prime number, and moreover, all these prime numbers are different. Here is the list of all the 40 prime numbers generated by Euler's polynomial:


Here we are concerned with the question whether there are still better quadratic polynomials in the sense that they generate even more prime numbers. We say that a quadratic polynomial $an^2 + bn + c$ has Euler quality $p$ if the $p$ numbers

$$|an^2 + bn + c|, \quad n = 0, \ldots, p - 1$$

are different prime numbers. By taking absolute values, we therefore also allow "negative primes". As an example, let us look at the polynomial $n^2 - 10n + 2$. For $n = 0$, we get 2 (prime), for $n = 1$ we obtain $-7$ (negative prime), and $n = 2$ gives $-14$ (no (negative) prime). The Euler quality of $n^2 - 10n + 2$ is therefore 2 but not higher.

Here is the challenge: write a program that systematically searches for a quadratic polynomial with high Euler quality. The goal is to find a polynomial with Euler quality larger than 40, in order to "beat" $n^2 + n + 41$. What is the highest Euler quality that you can find?

For this challenge, it can be useful to read the paragraph on constant expressions in the Details section.
Exercise 91 The XBM file format is a format for storing monochrome (black & white) images. The format is somewhat outdated, but many browsers (Internet Explorer is a notable exception) can still display images in XBM format.

An XBM image file for an image named test might look like this (taken from Wikipedia's XBM page).

```c
#define test_width 16
#define test_height 7
static unsigned char test_bits[] = {
  0x13, 0x00, 0x15, 0x00, 0x93, 0xcd, 0x55,
  0xa5, 0x93, 0xc5, 0x00, 0x80, 0x00, 0x60};
```

As you can guess from this, XBM files are designed to be integrated into C and C++ source code which makes it easy to process them (there is no need to read in the data; simply include the file from the C++ program that needs to process the image). In our example, test_width and test_height denote the width and height of the image in pixels. Formally, these names are macros, but in the program they can be used like constant expressions. test_bits is an array of characters that encodes the colors of the $16 \times 7$ pixels in the image. Every hexadecimal literal of the form $0x d_1 d_2$ encodes eight pixels, where the order is row by row. In our case, $0x13$ and $0x00$ encode the 16 pixels of the first row, while $0x15$ and $0x00$ are for the second row, etc.

Here is how a two-digit hexadecimal literal encodes the colors of eight consecutive pixels within a row. If the width is not a multiple of 8, the superfluous color values from the last hexadecimal literal of each row are being ignored.

Every hexadecimal digit $d_1$ is from the set \{0,...,9,a,...,f\} where a up to f stand for 10,...,15. The actual number encoded by a hexadecimal literal is $16d_1 + d_2 \in \{0,...,255\}$. If the type char has value range $\{-128,...,127\}$, the silent assumption is that a literal value $\alpha$ larger than 127 converts to $\alpha - 256$, which has the same representation under two's complement. For example, $0x13$ has value $1 \cdot 16 + 3 = 19$.

Now, any number in $\{0,...,255\}$ has a binary representation with 8 bits. 19, for example, has binary representation 00010011. The pixel colors are obtained by reading this backwards, and interpreting 1 as black and 0 as white. Thus, the first eight pixels in row 1 of the test image are black, black, white, white, black, white, white. The complete test image looks like this:

```
Blarg
```

Write a program xbm.cpp that #includes an XBM file of your choice (you may search the web to find suitable XBM files, or you may use an image manipulation program such as gimp to convert your favorite image into XBM format), and that outputs an XBM file for the same image, rotated by 90 degrees. The program may
write the resulting file to standard output. In case of the test image, the resulting XBM file and the resulting rotated image are as follows.

```c
#define rotated_width 7
#define rotated_height 16
static unsigned char rotated_bits[] =
  0x3c, 0x54, 0x48, 0x00, 0x04, 0x1c, 0x00, 0x1c,
  0x14, 0x08, 0x00, 0x1f, 0x00, 0x0a, 0x15, 0x1f};
```

Note that we now have 16 instead of 14 hexadecimal literals. This is due to the fact that each of the 16 rows needs one literal for its 7 pixels, where the leading bits of the binary representations are being ignored.

You may extend your program to perform other kinds of image processing tasks of your choice. Examples include color inversion (replace black with white, and vice versa), computing a mirror image, scaling the image (so that it occupies less or more pixels), etc.
Chapter 3

Functions
CHAPTER 3. FUNCTIONS

3.1 A first C++ function

Garbage in, garbage out.

Attributed to George Fuechsel, IBM, late 1950’s

This section introduces C++ functions as a means to encapsulate and reuse functionality, and to subdivide a program into subtasks. You will learn how to add functions to your programs, and how to call them. We also explain how functions can efficiently be made available for many programs at the same time, through separate compilation and libraries.

In many numerical calculations, computing powers is a fundamental operation (see Section 2.5), and there are many other operations that occur frequently in applications. In C++, functions are used to encapsulate such frequently used operations, making it easy to invoke them many times, with different arguments, and from different programs, but without having to reprogram them every time.

Even more importantly, functions are used to structure a program. In practice, large programs consist of many small functions, each of which serves a clearly defined subtask. This makes it a lot easier to read, understand, and maintain the program.

We have already seen quite a number of functions, since the main function of every C++ program is a special function (Section 2.1.4).

Program 18 emphasizes the encapsulation aspect and shows how functions can be used. It first defines a function for computing the value $b^e$ for a given real number $b$ and given integer $e$ (possibly negative). It then calls this function for several values of $b$ and $e$. The computations are performed over the floating point number type double.

```cpp
// Prog: callpow.cpp
// Define and call a function for computing powers.

#include <iostream>

// PRE: e >= 0 || b != 0.0
// POST: return value is b^e

double pow(double b, int e)
{
    double result = 1.0;
    if (e < 0) {
        // b^e = (1/b)^(-e)
        b = 1.0 / b;
        e = -e;
    }
    for (int i = 0; i < e; ++i) result *= b;
}
```
3.1. A FIRST C++ FUNCTION

```cpp
return result;
}

int main()
{
    std::cout << pow(2.0, -2) << "\n"; // outputs 0.25
    std::cout << pow(1.5, 2) << "\n"; // outputs 2.25
    std::cout << pow(5.0, 1) << "\n"; // outputs 5
    std::cout << pow(3.0, 4) << "\n"; // outputs 81
    std::cout << pow(-2.0, 9) << "\n"; // outputs -512

    return 0;
}
```

Program 18: ../progs/lecture/callpow.cpp

Before we explain the concepts necessary to understand this program in detail, let us get an overview of what is going on in the function `pow`. For nonnegative exponents e, \(b^e\) is obtained from the initial value of 1 by e-fold multiplication with b. This is what the for-loop does. The case of negative e can be handled by the formula \(b^e = (1/b)^{-e}\): after inverting b and negating e in the if-statement, we have an equivalent problem with a positive exponent. The latter only works if \(b \neq 0\), and indeed, negative powers of 0 are mathematically undefined.

### 3.1.1 Pre- and postconditions

Even a very simple function should document its precondition and its postcondition, in the form of comments. The precondition specifies what has to hold when the function is called, and the postcondition describes value and effect of the function. This information allows us to understand the function without looking at the actual source code; this in turn is a necessary for keeping track of larger programs. In case of the function `pow`, the precondition

```cpp
// PRE: e >= 0 || b != 0.0
```

tells us that e must be nonnegative, or (if e is negative) that \(b \neq 0\) must hold. The postcondition

```cpp
// POST: return value is b^e
```

tells us the function value, depending on the arguments. In this case, there is no effect.

The pre- and postconditions specify the function in a mathematical sense. At first sight, functions with values and effect do not fit into the framework of mathematical functions which only have values. But using the concept of program states (Section 1.2.3), a C++ function can be considered as a mathematical function that maps program states (immediately before the function call) to program states (immediately after the function call).
CHAPTER 3. FUNCTIONS

Under this point of view, the precondition specifies the domain of the function, the set of program states in which the function may be called. In case of pow, these are all program states in which the arguments \( b \) and \( e \) are in a suitable relation. The postcondition describes the function itself by specifying how the (relevant part of the) program state gets transformed. In case of \( \text{pow} \), the return value \( b^e \) will (temporarily) be put at some memory location.

To summarize, the postcondition tells us what happens when the precondition is satisfied. On the other hand, the postcondition gives no guarantee whatsoever for the case where the precondition is not satisfied. From a mathematical point of view, this is fine: a function is simply not defined for arguments outside its domain.

Arithmetic pre- and postconditions. The careful reader of Section 2.5 might have realized that both pre- and postcondition of the function \( \text{pow} \) cannot be correct. If \( e \) is too large, for example, the computation might overflow, but such \( e \) are not excluded by the precondition. Even if there is no overflow, the value range of the type double may have a hole at \( b^e \), meaning that this value cannot be returned by the function. The postcondition is therefore imprecise as well.

In the context of arithmetic operations over the fundamental C++ types, it is often tedious and even undesirable to write down precise pre- and postconditions; part of the problem is that fundamental types may behave differently on different platforms. Therefore, we often confine ourselves to pre- and postconditions that document our mathematical intention, but we have to keep in mind that in reality, the function might behave differently.

Assertions. So far, our preconditions are just comments like in

```cpp
// PRE: e >= 0 || b != 0.0
```

Therefore, if the function \( \text{pow} \) is called with arguments \( b \) and \( e \) that violate the precondition, this passes unnoticed. On the syntactical level, there is nothing we can do about it: the function call \( \text{pow} (0.0, -1) \), for example, will compile. But we can make sure that this blunder is detected at runtime. A simple way to do this uses assertions. An assertion has the form

```cpp
assert ( expr )
```

where \( expr \) is a predicate, an expression of a type whose values can be converted to bool. No comma is allowed in \( expr \), a consequence of the fact that \( \text{assert} \) is not a function but a macro. A macro is a piece of meta-code that the compiler replaces with actual C++ code prior to compilation.

With assertions, \( \text{pow} \) can be written as follows.

```cpp
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
```
3.1. A FIRST C++ FUNCTION

double pow (double b, int e)
{
    assert (e >= 0 || b != 0.0);
    double result = 1.0;
    // the remainder is as before
    ...
}

The purpose of an assertion is to check whether a certain predicate holds at a certain point. The precise semantics of an assertion is as follows. expr is evaluated, and if it returns false, execution of the program terminates immediately with an error message telling us that the respective assertion was violated. If expr returns true, execution continues normally. In our case, this means that the evaluation of the expression pow (0.0, -1) leads to a runtime error. This might not be a very polite way of telling the user that the arguments were illegal but the point will surely come across.

You can argue that it is costly to test the assertion in every function call, just to catch a few “bad” calls. However, it is possible to tell the compiler to ignore the assert macro, meaning that an empty piece of C++ code replaces it. The usual way to go is therefore as follows: during code development, put assertions everywhere you want to be sure that something holds. When the code is put into use, tell the compiler to remove the assertions. The machine language code is then as efficient as if you would never have written the assertions in the first place.

To use the assert macro, we have to include the header assert.h.

3.1.2 Function definitions

Lines 8–18 of Program 18 define a function called pow. The syntax of a function definition is as follows.

T fname ( T1 pname1, T2 pname2, ..., Tk pnamek )
block

This defines a function called fname, with return type T, and with formal arguments pname1, ..., pnamek of types T1, ..., Tk, respectively, and with a function body block.

Syntactically, T and T1, ..., Tk are type names, fname as well as pname1, ..., pnamek are identifiers (Section 2.1.10), and block is a block, a sequence of statements enclosed in curly braces (Section 2.4.3).

We can think of the formal arguments as placeholders for the actual arguments that are supplied (or “passed”) during a function call.

Function definitions must not appear inside blocks, other functions, or control statements. They may appear inside namespaces, though, or at global scope, like in callpow.cpp. A program may contain an arbitrary number of function definitions, appearing one af-
ter another without any delimiters between them. In fact, the program `callpow.cpp` consists of two function definitions, since the `main` function is a function as well.

### 3.1.3 Function calls

In Program [18] `pow(2.0,-2)` is one of five function calls. Formally, a function call is an expression. The syntax of a function call that matches the general function definition from above is as follows.

```
fname ( expr1, ..., exprk )
```

Here, `expr1,...,exprk` must be expressions of types whose values can be converted to the formal argument types `T1,...,Tk`. These expressions are the **call arguments**. For all types that we know so far, the call arguments as well as the function call itself are `rvalues`. The type of the function call is the function's return type `T`.

When a function call is evaluated, the call arguments are evaluated first (in an order that is unspecified by the C++ standard). The resulting values are then used to initialize the formal arguments. Finally, the function body is executed; in this execution, the formal arguments behave like they were variables defined in the beginning of block, initialized with the values of the call arguments.

In particular, if the formal arguments are of `const`-qualified type, they cannot be modified within the function body. For example, the following version of the function `pow` will not compile.

```cpp
double pow (const double b, const int e)
{
    double result = 1.0;
    if (e < 0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        // error: b is constant
        e = -e;
        // error: e is constant
    }
    for (int i = 0; i < e; ++i) result *= b;
    return result;
}
```

Const-correctness extends to formal function arguments: whenever the formal argument is meant to be kept constant within the function body, this should be documented by giving it a `const`-qualified type. From what we have said above about how function calls are evaluated, it follows that the "outside behavior" of the function (as documented by the pre- and postconditions) does not depend on whether the argument type is `const int` or `int` (see also Section [3.1.5] below). In this case, the effect of `const` is purely internal and determines whether the formal argument in question acts as a variable or as a constant within the function body.
For a fundamental type \( T \), there is no difference between return type \( T \) and return type \( \text{const} \ T \).

The evaluation of a function call terminates as soon as a return statement is reached, see Section 2.1.15. This return statement must be of the form

\[
\text{return expr;}
\]

where \( \text{expr} \) is an expression of a type whose values can be converted to the return type \( T \). The resulting value is the value of the function call. The effect of the function call is determined by the joint effects of the call argument evaluations, and of executing \textit{block}.

The function body may contain several return statements, but if no return statement is reached during the execution of \textit{block}, value and effect of the function call are undefined (unless the return type is \textit{void}, see Section 3.1.4 below).

For example, during the execution of \textit{block} in \texttt{pow(2.0,-2)}; \( b \) and \( e \) initially have values 2 and \(-2\). These values are changed in the if-statement to 0.5 and \( 2 \), before the subsequent loop sets result to 0.5 in its first and to 0.25 in its second (and last) iteration. This value is returned and becomes the value of the function call expression \texttt{pow(2.0,-2)}.

### 3.1.4 The type \textit{void}

In C++, there is a fundamental type called \textit{void}, used as return type for functions that only have an effect, but no value. Such functions are also called \textit{void functions}.

As an example, consider the following program (note that the function \texttt{print_pair} requires no precondition, since it works for every combination of int values).

```cpp
#include <iostream>

// POST: "(i, j)" has been written to standard output
void print_pair (const int i, const int j)
{
    std::cout << "( " << i << " , " << j << " )\n";
}

int main()
{
    print_pair(3,4); // outputs (3, 4)
}
```

The type \textit{void} has empty value range, and there are no literals, variables, or formal function arguments of type \textit{void}. There are expressions of type \textit{void}, though, for example \texttt{print_pair(3,4)}.

A \textit{void} function does not require a return statement, but it may contain return statements with \textit{expr} of type \textit{void}, or return statements of the form...
Evaluation of a void function call terminates when a return statement is reached, or when the execution of block is finished.

### 3.1.5 Functions and scope

The parenthesized part of a function definition contains the declarations of the formal arguments. For all of them, the declarative region is the function definition, so the formal arguments have local scope (Section 2.4.3). The potential scope of a formal argument declaration begins after the declaration and extends until the end of the function body. Therefore, the formal arguments are not visible outside the function definition. Within the body, the formal arguments behave like variables that are local to block.

In particular, changes made to the values of formal arguments (like in the function `pow`) are “lost” after the function call and have no effect on the values of the call arguments. This is not surprising, since the call arguments are rvalues, but to make the point clear, let us consider the following alternative main function in `callpow.cpp`.

```cpp
int main() {
    double b = 2.0;
    int e = -2;
    std::cout << pow(b, e); // outputs 0.25
    std::cout << b;       // outputs 2
    std::cout << e;       // outputs -2
    return 0;
}
```

The variables `b` and `e` defined in lines 2–3 are unaffected by the function call, since the function body of `pow` is not in the scope of their declarations, for two reasons. First, the definition of `pow` appears before the declarations of `b` and `e` in lines 2–3, so the body of `pow` cannot even be in the potential scope of these declarations. Second, even if we would move the declarations of the variables `b` and `e` to the beginning of the program (before the definition of `pow`, so that they have global scope), their scope would exclude the body of `pow`, since that body is in the potential scopes of redeclarations of the names `b` and `e` (the formal arguments), see Section 2.4.3.

But the general scope rules of Section 2.4.3 do allow function bodies to use names of global or namespace scope; the program on page 183 for example uses `std::cout` as such a name. Here is a contrived program that demonstrates how a program may modify a global variable (a variable whose declaration has global scope). While such constructions may be useful in certain cases, they usually make the program less readable, since the effect of a function call may then become very non-local.

```cpp
#include <iostream>
```
3.1. A FIRST C++ FUNCTION

```cpp
int i = 0; // global variable

void f()
{
  ++i; // in the scope of declaration in line 3
}

int main()
{
  f();
  std::cout << i << "\n"; // outputs 1
  return 0;
}
```

Since the formal arguments of a function have local scope, they also have automatic storage duration. This means that we get a "fresh" set of formal arguments every time the function is called, with memory assigned to them only until the respective function call terminates.

**Function declarations.** A function itself also has a scope, and the function can only be called within its scope. The scope of a function is obtained by combining the scopes of all its declarations. The part of the function definition before block is a declaration, but there may be function declarations that have no subsequent block. This is in contrast to variables where every declaration is at the same time a definition. A function may be declared several times, but it can be defined once only.

The following program, for example, does not compile, since the call of f in main is not in the scope of f.

```cpp
#include <iostream>

int main()
{
  std::cout << f(1); // f undeclared
  return 0;
}
```

```cpp
int f(const int i) // scope of f begins here
{
  return i;
}
```

But we can put f into the scope of main by adding a declaration before main, and this yields a valid program.

```cpp
#include <iostream>
```
int f (int i); // scope of f begins here

int main()
{
    std::cout << f(1); // ok, call is in scope of f
    return 0;
}

int f (const int i)
{
    return i;
}

In declarations, we omit const-qualifications of arguments of fundamental type. As explained in Section 3.1.3 such qualifications are useful only internally, but the declaration is about the outside behavior of the function. For the compiler, the two declarations

int f (int i);

and

int f (const int i);

are identical.

In the previous program, we could get rid of the extra declaration by simply defining f before main, but sometimes, separate function declarations are indeed necessary. Consider two functions f and g such that g is called in the function body of f, and f is called in the function body of g. We have to define one of the two functions first (f, say), but since we call g within the body of f, g must have a declaration before the definition of f.

3.1.6 Procedural programming

So far, we have been able to “live” without functions only because the programs that we have written are pretty simple. But even some of these simple ones would benefit from functions. Consider as an example the program perfect.cpp from Exercise 49. In this exercise, we have asked you to find the perfect numbers between 1 and n, for a given input number n. The solution so far uses one “big” double loop (loop within a loop) that in turn contains two if statements. Although in this case, the “big” loop is still small enough to be read without difficulties, it doesn’t really reflect the logical structure of the solution. Once we get to triple or quadruple loops, the program may become very hard to follow.

But what is the logical structure of the solution? For every number i between 1 and n, we have to test whether i is perfect; and to do the latter, we have to compute the sum of all proper divisors of i and check whether it is equal to i. Thus, we have two clearly defined subtasks that the program has to solve for every number i, and it is best to encapsulate
3.1. A FIRST C++ FUNCTION

these into functions. Program [19] shows how this is done. Note that the program is now almost self-explanatory: the postconditions can more or less directly be read off the function names.

```cpp
// Program: perfect2.cpp
// Find all perfect numbers up to an input number n

#include <iostream>

// POST: return value is the sum of all divisors of i
//       that are smaller than i
unsigned int sum_of_proper_divisors(const unsigned int i) {
    unsigned int sum = 0;
    for (unsigned int d = 1; d < i; ++d)
        if (i % d == 0) sum += d;
    return sum;
}

// POST: return value is true if and only if i is a perfect number
bool is_perfect(const unsigned int i) {
    return sum_of_proper_divisors(i) == i;
}

int main() {
    // input
    std::cout << "Find perfect numbers up to n =? ";
    unsigned int n;
    std::cin >> n;

    // computation and output
    std::cout << "The following numbers are perfect.\n";
    for (unsigned int i = 1; i <= n; ++i)
        if (is_perfect(i)) std::cout << i << " ";
    std::cout << "\n"
    return 0;
}
```

Program 19: ../progs/lecture/perfect2.cpp

Admittedly, the program is longer than perfect.cpp, but it is more readable, and it has simpler control flow. In particular, the double loop has disappeared.
The larger a program gets, the more important is it to subdivide it into small subtasks, in order not to lose track of what is going on in the program on the whole; this is the *procedural programming* paradigm, and in C++, it is realized with functions.

The procedural programming paradigm is not so self-evident as it may seem today. The first programming language that became accessible to a general audience since the 1960's was BASIC (Beginner's All-purpose Symbolic Instruction Code).

In BASIC, there were no functions; in order to execute a code fragment responsible for a subtask, you had to use the GOTO statement (with a line number)—or GOSUB in many dialects—to jump to that code, and then jump back using another GOTO (RETURN, respectively). The result was often referred to as *spaghetti code*, due to the control flow meandering like a boiled spaghetti on a plate. Moreover, programmers often didn't think in terms of clearly defined subtasks, simply because the language did not support it. This usually lowered the code quality even further.

Despite this, BASIC was an extremely successful programming language. It reached the peak of its popularity in the late 1970's and early 1980's when the proud owners of the first home computers (among them the authors) created programs of fairly high complexity in BASIC.

### 3.1.7 Arrays as function arguments

We have seen in Section 2.6.2 that an array cannot be initialized from another array, and this implies that arrays have to receive special attention in the context of functions. The usual first step in a function call evaluation (the call arguments are evaluated, and their values are used to initialize the formal arguments) won't work with arrays.

Given this, it might be surprising that formal arguments of array type are allowed. For example, we could declare a function

```c
// PRE: a[0],...,a[n-1] are elements of an array
// POST: a[i] is set to value, for 0 <= i < n
void fill_n ( int a[], int n, int value);
```

to set all elements of an array to some fixed value. The compiler, however, internally adjusts this to the completely equivalent declaration

```c
// PRE: a[0],...,a[n-1] are elements of an array
// POST: a[i] is set to value, for 0 <= i < n
void fill_n ( int* a, int n, int value);
```

The same adjustment would happen for the formal argument `int a[5]`, say, meaning that the array length is ignored. You could in fact (legally, but quite confusingly) have a formal argument `int a[5]`, and then use an array of length 10 as call argument.

The moral is that in reality, no formal arguments of array type exist, and in order to avoid confusion, it is better not to pretend otherwise.

If we want to build a function that works with arrays, we therefore have to think about alternative ways of passing the array to the function. An obvious way is suggested
by the declaration of fill_n above: we pass a pointer to the first element, along with
the number of elements. But the standard way is to pass two pointers, one to the
first element, and a past-the-end pointer. This also uniquely describes the array. We
may actually choose the call arguments in such a way that they describe a contiguous
subarray of the original array. This generalization is possible since the array itself never
appears as an argument.

Here is a program that defines and uses a fill function in the standard way (which is
now just called fill since there is no n).

```
1 // Program: fill.cpp
2 // define and use a function to fill an array
3
4 #include <iostream>
5
6 // PRE: [first, last) is a valid range
7 // POST: *p is set to value, for p in [first, last)
8 void fill (int* first, int* last, const int value) {
9   // iteration by pointer
10   for (int* p = first; p != last; ++p)
11     *p = value;
12 }
13
14 int main()
15 {
16   int a[5];
17   fill (a, a+5, 1); // a == {1, 1, 1, 1, 1}
18   return 0;
19 }
```

Program 20: ../progs/lecture/fill.cpp

In fill, we iterate over all addresses in the range [first, last), and we get the array
elements by dereferencing. A valid range contains the addresses of a (possibly empty)
set of consecutive array elements, where the halfopen interval notation [first, last)
means that the range is given by the values of first, first+1, ..., last-1. In other
words, last is a past-the-end pointer for the subarray described by the range.

Let us make a small digression and try to understand why the "standard way" of
passing an array as a range (as in fill) is preferable to passing it as a pointer to its first
element plus its length (as envisioned in the declaration of fill_n). In C++, there are
techniques to make functions like fill or fill_n available not only for arrays, but for
many other containers at the same time. In this general setting, the functions work with
iterators.

As already indicated on page 147 in Section 2.6.9 we may think of iterators as gen-
eralized pointers to container elements, but the operations that we can actually perform
on these "pointers" depend on the container. The way we have defined a container in
Section 2.6.5 we are guaranteed that we can iterate over its elements. By convention, this is realized through “pointer” increment, using the operator ++. Therefore, the function fill as above has the potential to work for all containers, since it only requires “pointer” functionality that is offered by all container iterators.

Now, if you think about how to implement fill_n using only the operator ++ on pointers, you realize that you need an extra counting variable to find out when you are done. You cannot simply construct a past-the-end iterator a+n, since this requires random access functionality that the container in question may not offer. Thus, you either end up with a more complicated, or with a less general fill function than fill. There are some situations where array-passing as in fill_n is appropriate, but in the majority of the cases, you want to do it as in fill.

Mutating functions. There is a substantial difference between the function pow on the one hand, and the function fill on the other hand. A call to the function pow has no effect, since the computations only modify formal argument values; these values are “local” to the function call and are lost upon termination. With pow, it’s the value of a function call that we are interested in.

Calls to the function fill, on the other hand, have effects: they modify the values of array elements, and these values are not local to the function call. When we write

```c
int a[5];
fill (a, a+5, 0);
```

the effect of the expression fill (a, a+5, 0) is that all elements of a receive value 0. This is possible since there are formal arguments of pointer type. When the function call fill (a, a+5, 0) is evaluated, the formal argument first is initialized with the address of a’s first element. In the function body, the value at this address is modified through the lvalue *p, and the same happens for the other four array elements in turn.

Formal arguments of pointer type are therefore a means of constructing functions with value-modifying effects. Such functions are called mutating.

Of course, there are also functions on arrays that are non-mutating, and in this case, const-correctness requires us to document this in the types of the formal arguments. As an example, consider the following function for finding the minimum element in a nonempty range of integers delimited by two pointers.

```c
// PRE: [first, last) is a valid nonempty range
// POST: the smallest of the values described by [first, last) is returned
int min_element (const int* first, const int* last) {
    assert (first != last);
    int m = *first++;
    // current candidate for the minimum
    for (const int *p = first; p < last; ++p)
        if (*p < m) m = *p;
    return m;
}
```
As the function \texttt{min_element} does not intend to modify the objects described by the range \texttt{[first, last)}, the respective pointers have const-qualified underlying type \texttt{const int} (see also Section \ref{sec:2.6.8}). For all formal arguments of \texttt{non-pointer} type that we know so far, the \texttt{const} keyword makes no difference outside the function body, since the values of the call arguments cannot be changed by the function anyway. For formal arguments of \texttt{pointer} type, the difference is essential and lets us distinguish mutating from non-mutating functions.

### 3.1.8 Modularization

There are functions that are tailor-made for a specific program, and it would not make sense to use them in another program. But there are also general purpose functions that are useful in many programs. It is clearly undesirable to copy the corresponding function definition into every program that calls the function; what we need is \textit{modularization}, a subdivision of the program into independent parts.

The power function \texttt{pow} from Program \ref{sec:18} is certainly general purpose. In order to make it available to all our programs, we can simply put the function definition into a separate source code file \texttt{pow.cpp}, say, in our working directory.

---

```cpp
#include <cassert>

// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
do double pow (double b, int e)
{
    assert (e >= 0 || b != 0.0);
    double result = 1.0;
    if (e < 0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i=0; i<e; ++i) result *= b;
    return result;
}
```

---

Program 21: ../progs/lecture/pow.cpp

Then we can include this file from our main program as follows.

```cpp
#include <cassert>

// Prog: callpow2.cpp
// Call a function for computing powers.
```
# include <iostream>
# include "pow.cpp"

int main()
{
    std::cout << pow(2.0, -2) << "\n"; // outputs 0.25
    std::cout << pow(1.5, 2) << "\n"; // outputs 2.25
    std::cout << pow(5.0, 1) << "\n"; // outputs 5
    std::cout << pow(3.0, 4) << "\n"; // outputs 81
    std::cout << pow(-2.0, 9) << "\n"; // outputs -512
    return 0;
}

Program 22: ../progs/lecture/callpow2.cpp

An include directive of the form

```
#include "filename"
```

logically replaces the include directive by the contents of the specified file. Usually, filename is interpreted relative to the working directory.

Separate compilation and object code files. The code separation mechanism from the previous paragraph has one major drawback: the compiler does not “see” it. Before compilation, pow.cpp is logically copied back into the main file, so the compiler still has to translate the function definition into machine language every time it compiles a program that calls pow. This is a waste of time that can be avoided by separate compilation.

In our case, we would compile the file pow.cpp separately. We only have to tell the compiler that it should not generate an executable program (it can’t, since there is no main function) but an object code file, called pow.o, say. This file contains the machine language instructions that correspond to the C++ statements in the function body of pow.

Header files. The separate compilation concept is more powerful than we have seen so far: surprisingly, even programs that call the function pow can be compiled separately, without knowing about the source code file pow.cpp or the object code file pow.o. What the compiler needs to have, though, is a declaration of the function pow.

This function declaration is best put into a separate file as well. In our case, this file pow.h, say, is very short; it contains just the lines

```
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
double pow (double b, int e);
```
Since this is the “header” of the function pow, the file pow.h is called a \textit{header file}. In the calling Program\ref{callpow2} we simply replace the inclusion of pow.cpp by the inclusion of pow.h, resulting in the following program.

```
1  // Prog: callpow3.cpp
2  // Call a function for computing powers.
3
4  #include <iostream>
5  #include "pow.h"
6
7  int main()
8  {
9    std::cout << pow( 2.0, -2) << "\n"; // outputs 0.25
10   std::cout << pow( 1.5,  2) << "\n"; // outputs 2.25
11   std::cout << pow( 5.0,  1) << "\n"; // outputs 5
12   std::cout << pow( 3.0,  4) << "\n"; // outputs 81
13   std::cout << pow(-2.0,  9) << "\n"; // outputs -512
14
15   return 0;
16 }
```

\textbf{Program 23: ../progs/lecture/callpow3.cpp}

From this program, the compiler can then generate an object code file callpow3.o. Instead of the machine language instructions for executing the body of pow, this object code contains a \textit{placeholder} for the location under which these instructions are to be found in the executable program. It is important to understand that callpow3.o cannot be an executable program yet: it does contain machine language code for main, but not for another function that it needs, namely pow.

\begin{itemize}
\item \textbf{The linker.} Only when an executable program is built from callpow3.o, the object code file pow.o comes into play. Given all object files that are involved, the \textit{linker} builds the executable program by gluing together machine language code for function calls (in callpow3.o) with machine language code for the corresponding function bodies (in pow.o). Technically, this is done by putting all object files together into a single executable file, and by filling placeholders for function body locations with the actual locations in the executable.
\item Separate compilation is very useful. It allows to change the definition of a function without having to recompile a single program that calls it. As long as the function declaration remains unchanged, it is only the linker that has to work in the end; and the linker is usually very fast. It follows that separate compilation also makes sense for functions that are specific to one program only.
\item Separate compilation reflects the “customer” view of the calling program: as long as a function does what its pre- and postcondition promise in the header file, it is not
important to know how it does this. On the other hand, if the function definition is hidden from the calling program, clean pre- and postconditions are of critical importance, since they may be the only information available about the function’s behavior.

Availability of sourcecode. If you have carefully gone through what we have done so far, you realize that we could in principle delete the sourcecode file `pow.cpp` after having generated `pow.o`, since later, the function definition is not needed anymore. When you buy commercial software, you are often faced with the absence of sourcecode files, since the vendor does not want customers to modify the sourcecode instead of buying updates, or to discover how much money they have paid for lousy software. (To be fair, we want to remark that there are also more honest reasons for not giving away sourcecode.)

In academic software, availability of sourcecode goes without saying. In order to evaluate or reproduce the contribution of such software to the respective area of research, it is necessary to have sourcecode. Even in commercial contexts, open source software is advancing. The most prominent software that comes with all sourcecode files is the operating system Linux. Open source software can very efficiently be adapted and improved if many people contribute. But such a contribution is possible only when the sourcecode is available.

Libraries. The function `pow` will not be the only mathematical function that we want to use in our programs. To make the addition of new functions easy, we can put the definition of `pow` (and similar functions that we may add later) into a single sourcecode file `math.cpp`, say, and the corresponding declarations into a single header file `math.h`. The object code file `math.o` then contains machine language code for all our mathematical functions.

Although not strictly necessary, it is good practice to include `math.h` in the beginning of `math.cpp`. This ensures consistency between function declarations and function definitions and puts the code in `math.cpp` into the scope of all functions declared in `math.h`, see Section 3.1.5. In all function bodies in `math.cpp`, we can therefore call the other functions, without having to think about whether these functions have already been declared.

In general, several object code files may be needed to generate an executable program, and it would be cumbersome to tell the linker about all of them. Instead, object code files that logically belong together can be archived into a library. Only the name of this library must then be given to the linker in order to have all library functions available for the executable program. In our case, we so far have only one object file `math.o` resulting from `math.cpp`, but we can still build a library file `libmath.a`, say, from it.

Figure 18 schematically shows how object code files, a library and finally an executable program are obtained from a number of sourcecode files.

Centralization and namespaces. It is clear that we do not want to keep header files and libraries of general interest in our working directory, since we (and others) may have
3.1. A FIRST C++ FUNCTION

Figure 18: Building object code files, libraries and executable programs.

many working directories. Header files and libraries should be at some central place.

We can make our programs independent from the location of header files by writing

```
#include <filename>
```

but in this case, we have to tell the compiler (when we start it) where to search for files to be included. This is exactly the way that headers like `iostream` from the standard library are included; their locations are known to the compiler, so we don’t have to provide any information here. Similarly, we can tell the linker where the libraries we need are to be found. Again, for the various libraries of the standard library, the compiler knows this. We want to remark that `filename` is not necessarily the name of a physical file; the mapping of `filename` to actual files is implementation defined.

Finally, it is good practice to put all functions of a library into a namespace, in order to avoid clashes with user-declared names, see Section 2.1.3. Let us use the namespace `ifm` here.

Here are the header and implementation files `math.h` and `math.cpp` that result from these guidelines for our intended library of mathematical functions (that currently contains `pow` only).

```
1 // math.h
2 // A small library of mathematical functions.
```
CHAPTER 3. FUNCTIONS

namespace ifm {
   // PRE: e >= 0 || b != 0.0
   // POST: return value is b^e
   double pow (double b, int e);
}

Program 24: ../progs/lecture/math.h

#include <cassert>
#include <IFM/math.h>

namespace ifm {
   double pow (double b, int e) {
      assert (e >= 0 || b != 0.0);
      // PRE: e >= 0 || b != 0.0
      // POST: return value is b^e
      double result = 1.0;
      if (e < 0) {
         // b^e = (1/b)^(-e)
         b = 1.0/b;
         e = -e;
      }
      for (int i=0; i<e; ++i) result *= b;
      return result;
   }
}

Program 25: ../progs/lecture/math.cpp

Finally, the program callpow4.cpp calls our library function ifm::pow. It includes the header file math.h from a central directory IFM.
### 3.1. A FIRST C++ FUNCTION

```cpp
8 {
9   std::cout << ifm::pow(2.0, -2) << "\n"; // outputs 0.25
10  std::cout << ifm::pow(1.5, 2) << "\n"; // outputs 2.25
11  std::cout << ifm::pow(5.0, 1) << "\n"; // outputs 5
12  std::cout << ifm::pow(3.0, 4) << "\n"; // outputs 81
13  std::cout << ifm::pow(-2.0, 9) << "\n"; // outputs -512
14  return 0;
15 }
```

**Program 26: */progs/lecture/callpow4.cpp*

### 3.1.9 Using library functions

You can imagine that we were not the first to put a function like `pow` into a library. Indeed, the standard library contains a function `std::pow` that is even more general than ours: it can compute $b^e$ for real exponents $e$. Accordingly, the arguments of `std::pow` and its return value are of type `double`. In order to use this function, we have to include the header `cmath`. This header contains declarations for a variety of other numerical functions.

Using functions from the standard library can help us to get shorter, better, or more efficient code, without having to write a single new line by ourselves. For example, computing square roots can speed up our primality test in Program 8. You might have realized this much earlier, but when we are looking for some proper divisor of a natural number $n \geq 2$, it is sufficient to search in the range $\{2, \ldots, \lfloor \sqrt{n} \rfloor\}$. Indeed, if $n$ can be written as a product $n = dd'$, then the smaller of $d$ and $d'$ must be bounded by $\sqrt{n}$; since the divisors are integral, we even get a bound of $\lfloor \sqrt{n} \rfloor$, $\sqrt{n}$ rounded down.

The primality test could therefore be written more efficiently; Program 27 does this, using the function `std::sqrt` from the library `cmath`, whose argument and return types are `double`.

```cpp
1  // Program: prime2.cpp
2  // Test if a given natural number is prime.
3
4 #include <iostream>
5 #include <cmath>
6
7 int main ()
8 {
9   // Input
10   unsigned int n;
11   std::cout << "Test if n>1 is prime for n =? ";
12   std::cin >> n;
13 ```
CHAPTER 3. FUNCTIONS

14 // Computation: test possible divisors d up to sqrt(n)
15 const unsigned int bound = (unsigned int)(std::sqrt(n));
16 unsigned int d;
17 for (d = 2; d <= bound && n % d != 0; ++d);
18
19 // Output
20 if (d <= bound)
21 // d is a divisor of n in {2,...,[sqrt(n)]}
22 std::cout << n << " = " << d << " * " << n / d << ".\n";
23 else
24 // no proper divisor found
25 std::cout << n << " is prime.\n";
26
27 return 0;
28

Program 27: ../progs/lecture/prime2.cpp

The program is correct: if d <= bound still holds after the loop, we have left the loop because the other condition n % d != 0 has failed. This means that we have found a divisor. If d > bound holds after the loop, we have tried all possible divisors smaller or equal to bound (whose value is \[\sqrt{n}\]), since the explicit conversion rounds down, see Section 2.5.3), so we certainly have not missed any divisor. But we have to be a little careful here: our arguments assume that std::sqrt works correctly for squares. For example, std::sqrt(121) must return 11 (a little more wouldn't hurt), but not 10.99998, say. In that latter case, (unsigned int)(std::sqrt(121)) would have value 10, and by making this our bound, we miss the divisor 11 of 121, erroneously concluding that 121 is prime.

It is generally not safe to rely on some precise semantics of library functions, even if your platform implements floating point arithmetic according to the IEEE standard 754 (see Section 2.5.6). The square root function is special in the sense that the IEEE standard still guarantees the result of std::sqrt to be the floating point number closest to the real square root; consequently, our above implementation of the primality test is safe. But similar guarantees do not necessarily hold for other library functions.

Also in our second prime number application, Eratosthenes’s Sieve, we’d better call a standard library function in order to initialize our list of crossed out numbers, instead of doing it ourselves with a loop. For this, we would replace the two lines

for (unsigned int i = 0; i < n; ++i)
crossed_out[i] = false;

of Program 15 with the single line

std::fill (crossed_out, crossed_out + n, false);

The pre- and postconditions of this standard library function exactly match the ones of our own fill function from Page 189. The benefit here is not the saving of one line of
3.1. A FIRST C++ FUNCTION

code; this saving does not even exist, since we additionally have to \#include <algorithm>
in the beginning of the program.

The benefit is that we eliminate possible sources of error (even a trivial loop has the
potential of being wrong), and that we simplify the control flow (see also Section 2.4.8).

3.1.10 Details

Default arguments. Some functions have the property that there are “natural” values for
one or more of their formal arguments. For example, when filling an array of underlying
type int, the value 0 is such a natural value. In such a case, it is possible to specify
this value as a default argument; this allows the caller of the function to omit the
corresponding call argument and let the compiler insert the default value instead. In
case of the function fill from Program 20, this would look as follows.

```
// PRE: [first, last) is a valid range
// POST: *p is set to value, for p in [first, last)
void fill (int* first, int* last, const int value = 0) {
    // iteration by pointer
    for (int* p = first; p != last; ++p)
        *p = value;
}
```

This function can now be called with either two or three arguments, as follows.

```
int a[5];
fill (a, a+5);  // means: fill (a, a+5, 0)
fill (a, a+5, 1);
```

In general, there can be default values for any number of formal arguments, but these
arguments must be at consecutive positions i, i+1, ..., k among the k arguments, for
some i. The function can then be called with any number of call arguments between
i−1 and k, and the compiler automatically inserts the default values for the missing call
arguments.

A function may have a separate declaration that specifies default arguments, like in
the following declaration of fill.

```
// PRE: [first, last) is a valid range
// POST: *p is set to value, for p in [first, last)
void fill (int* first, int* last, int value = 0);
```

In this case, the actual definition must not repeat the default arguments (the actual rules
are a bit more complicated, but this is the upshot).

Function signatures. In function declarations, the formal argument names pname1, ..., 
pnamek can be omitted.

This makes sense since these names are only needed in the function definition. The
important information, namely domain and range of the function, are already specified
by the argument types and the return type. All these types together form the signature of the function.

In math.h, we could therefore equivalently write the declaration

double pow (double, int);

The only problem is that we need the formal argument names to specify pre- and post-conditions, without going to lengthy formulations involving “the first argument” and “the second argument”. Therefore, we usually write the formal argument names even in function declarations.

Mathematical functions. Many of the mathematical functions that are available on scientific pocket calculators are also available from the math library cmath. The following table lists some of them. All are available for the three floating point number types float, double and long double.

<table>
<thead>
<tr>
<th>name</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>std::abs</td>
<td>(</td>
</tr>
<tr>
<td>std::sin</td>
<td>( \sin(x) )</td>
</tr>
<tr>
<td>std::cos</td>
<td>( \cos(x) )</td>
</tr>
<tr>
<td>std::tan</td>
<td>( \tan(x) )</td>
</tr>
<tr>
<td>std::asin</td>
<td>( \sin^{-1}(x) )</td>
</tr>
<tr>
<td>std::acos</td>
<td>( \cos^{-1}(x) )</td>
</tr>
<tr>
<td>std::atan</td>
<td>( \tan^{-1}(x) )</td>
</tr>
<tr>
<td>std::exp</td>
<td>( e^x )</td>
</tr>
<tr>
<td>std::log</td>
<td>( \ln x )</td>
</tr>
<tr>
<td>std::log10</td>
<td>( \log_{10} x )</td>
</tr>
<tr>
<td>std::sqrt</td>
<td>( \sqrt{x} )</td>
</tr>
</tbody>
</table>

3.1.11 Goals

Dispositional. At this point, you should . . .

1) be able to explain the purpose of functions in C++;
2) understand the syntax and semantics of C++ function definitions and declarations;
3) know what the term “procedural programming” means;
4) understand the function pow from Program 18 and the functions fill_n and fill from Program 20;
5) know that formal arguments of pointer type can be used to write array-processing functions, and mutating functions;
6) know why it makes sense to compile function definitions separately, and to put functions into libraries.
3.1. A FIRST C++ FUNCTION

Operational. In particular, you should be able to . . .

(G1) give two reasons why it is desirable to subdivide programs into functions;

(G2) find pre- and postconditions for given functions, where the preconditions should
be as weak as possible, and the postconditions should be as strong as possible;

(G3) find syntactical and semantical errors in function definitions, and in programs
that contain function definitions;

(G4) evaluate given function call expressions;

(G5) understand and write functions that perform simple arrayprocessing tasks;

(G6) write (mutating) functions for given tasks, and write programs for given tasks
that use functions, observing const-correctness;

(G7) subdivide a given task into small subtasks, and write a program for the given task
that uses functions to realize the subtasks;

(G8) build a library on your platform, given that you are told the necessary technical
details.

3.1.12 Exercises

Exercise 92 Find pre- and postconditions for the following functions. (G2)(G4)

a) double f (const double i, const double j, const double k)
   
   if (i > j)
   if (i > k)
      return i;
   else
      return k;
   else
   if (j > k)
      return j;
   else
      return k;

b) double g (const int i, const int j)
   
   double r = 0.0;
   for (int k = i; k <= j; ++k)
      r += 1.0 / k;
   return r;
Exercise 93 What are the problems (if any) with the following functions? Fix them and find appropriate pre- and postconditions.

a) `bool is_even (const int i)`
   
   ```cpp
   { 
     if (i % 2 == 0) return true; 
   }
   ```

b) `double inverse (const double x)`
   
   ```cpp
   { 
     double result; 
     if (x != 0.0) 
       result = 1.0 / x; 
     return result; 
   }
   ```

Exercise 94 What is the output of the following program, depending on the input number `i`? Describe the output in mathematical terms, ignoring possible over- and underflows.

```cpp
#include <iostream>

int f (const int i)
{ 
  return i * i; 
}

int g (const int i)
{ 
  return i * f(i) * f(f(i)); 
}

void h (const int i)
{ 
  std::cout << g(i) << "\n"; 
}

int main()
{ 
  int i; 
  std::cin >> i; 
  h(i); 

  return 0; 
}
```
Exercise 95 Find three problems in the following program.

```cpp
#include <iostream>

double f (const double x)
{
    return g(2.0 * x);
}

bool g (const double x)
{
    return x % 2.0 == 0;
}

void h ()
{
    std::cout << result;
}

int main()
{
    const double result = f(3.0);
    h();

    return 0;
}
```

Exercise 96 Simplify the program from Exercise 91 by using the library function `std::pow`.

Exercise 97 Assume that on your platform, the library function `std::sqrt` is not very reliable. For a value of type `double` (x ≥ 0), we let s(x) be the value returned by `std::sqrt(expr)`, if expr has value x, and we assume that we only know that for some positive value $\varepsilon \leq 1/2$, the relative error satisfies

$$\frac{|s(x) - \sqrt{x}|}{\sqrt{x}} \leq \varepsilon, \quad \forall x.$$

How can you change Program 27 such that it correctly works under this relative error bound? You may assume that the floating point number system used on your platform is binary, and that all values of type `unsigned int` are exactly representable in this system. (This is a theory exercise.)

Exercise 98
a) Write a function

    // POST: return value is true if and only if n is prime
    bool is_prime (unsigned int n);

    and use this function in a program to count the number of twin primes in the
    range \{2, \ldots, 10000000\} (two up to ten millions). A twin prime is a pair of
    numbers \(i, i+2\) both of which are prime.

b) Is the approach of a) the best (most efficient) one to this problem? If you
    can think of a better approach, you are free to implement it instead of the one
    outlined in a).

Exercise 99 The function pow in Program 18 needs \(|e|\) multiplications to compute \(b^e\). Change the function body such that less multiplications are performed. You may
use the following fact. If \(e \geq 0\) and \(e\) has binary representation

\[
e = \sum_{i=0}^{\infty} b_i2^i,
\]

then

\[
b^e = \prod_{i=0}^{\infty} (b^{2^i})^{b_i}.
\]

Exercise 100 Write a program swap.cpp that defines and calls a function for inter-
changing the values of two int objects. The program should have the following
structure.

```
#include <iostream>

// your function definition goes here

int main() {
    // input
    std::cout << "i =? ";
    int i; std::cin >> i;

    std::cout << "j =? ";
    int j; std::cin >> j;

    // your function call goes here
```
3.1. A FIRST C++ FUNCTION

```cpp
// output
std::cout << "Values after swapping: i = " << i
<< ", j = " << j << ".\n";

return 0;
}

Here is an example run of the completed program:
i =? 5
j =? 8
Values after swapping: i = 8, j = 5.
```

Exercise 101 Write a program unique.cpp that implements and tests the following function.

// PRE: [first, last) is a valid range and describes a sequence
// of elements that are sorted in nondecreasing order
// POST: the return value is true if and only if no element
// occurs twice in the sequence
bool unique (const int* first, const int* last);

Exercise 102 Modify the program sort_array.cpp from Exercise 83 in such way that the resulting program sort_array2.cpp defines and calls a function

// PRE: [first, last) is a valid range
// POST: the elements *p, p in [first, last) are
// in ascending order
void sort (int* first, int* last);

to perform the sorting of the array into ascending order. It may be tempting (but not allowed for obvious reasons) to use std::sort or similar standard library functions in the body of the function sort that is to be defined. It is allowed, though, to compare the efficiency of your sort function with that of std::sort (which has the same pre- and postconditions and can be used after include<algorithm>). For this exercise, it is desirable (but not strictly necessary) to use pointer increment (++p) as the only operation on pointers (apart from initialization and assignment, of course). If you succeed in doing so, your sorting function has the potential of working for containers that do not offer random access (see also Section 3.1.7).

Exercise 103 Provide the postcondition for the following function. The postcondition must completely describe the behavior of the function for all valid inputs.
a) // PRE: [b, e) and [o, o+(e-b)) are disjoint valid ranges
void f (int* b, int* e, int* o)
{
    while (b != e) *(o++) = *(--e);
}

b) Which of the three following function calls are valid according to the precondition?
int a[5] = {1,2,3,4,5};

f(a, a+5, a+5);
f(a, a+2, a+3);
f(a, a+3, a+2);

c) Is the function f implemented in a const-correct fashion? If not, where are const’s
to be added?

Exercise 104 What does the following function do if e - b has value 5? To answer
this, write down the values of b[0], b[1], . . . , b[4] after a call to f(b, b+5).
// PRE: [b, e) is a valid range
void f(unsigned int* b, unsigned int* e)
{
    int n = e - b;
    for (int i = 0; i < n; ++i) {
        b[i] = 1;
        for (int j = i - 1; j > 0; --j)
            b[j] += b[j - 1];
    }
}

Can you describe the behavior (and thus provide a postcondition) for general
value of e-b?

Exercise 105 A perpetual calendar can be used to determine the weekday (Monday,
. . . , Sunday) of any given date. You may for example know that the Berlin wall
came down on November 9, 1989, but what was the weekday? (It was a Thursday.)
Or what is the weekday of the 1000th anniversary of the Swiss confederation, to be
celebrated on August 1, 2291? (Quite adequately, it will be a Saturday.)

a) The task is to write a program that outputs the weekday (Monday, . . . , Sun-
day) of a given input date.

Identify a set of subtasks to which you can reduce this task. Such a set is
not unique, of course, but all individual subtasks should be small (so small
that they could be realized with few lines of code). It is of course possible
for a subtask in your set to reduce to other subtasks. (Without giving away
3.1. A FIRST C++ FUNCTION

Anything, one subtask that you certainly need is to determine whether a given
year is a leap year).

b) Write a program perpetual_calendar.cpp that reads a date from the input and
outputs the corresponding weekday. The range of dates that the program can
process should start no later than January 1, 1900 (Monday). The program
should check whether the input is a legal date, and if not, reject it. An example
run of the program might look like this.

day =? 13
month =? 11
year =? 2007
Tuesday

To structure your program, implement the subtasks from a) as functions, and
put the program together from these functions.

Exercise 106 Build a library on your platform from the files math.h and math.cpp in
Program 24 and Program 25. Use this library to generate an executable program
from Program 26.

Exercise 107

a) Implement the following function and test it. You may assume that the type
double complies with the IEEE standard 754, see Section 2.5.6. The function
is only required to work correctly, if the nearest integer is in the value range
of the type int.

// POST: return value is the integer nearest to x
int round (double x);

b) The postcondition of the function does not say what happens if there are two
nearest integers. Specify the behavior of your implementation in the postcon-
dition of your function.

c) Add a declaration of your function to the file math.h (Program 24) and a definition
to math.cpp (Program 25). Build a library from these two files, and rewrite your
test function from a) to call the library version of the function round.

Exercise 108 This is another (not too difficult) one from Project Euler (Problem 56,
http://projecteuler.net/). Find natural numbers a,b < 100 for which a^b has the
largest cross sum (sum of decimal digits). Let us say upfront that 99^99 is not the
answer.

Write a program power_cross_sums.cpp that computes the best a and b (within
reasonable time).

Can you also find the best a, b up to 1,000?
Exercise 109 (This is a theory challenge.) The simplest computer model that is being studied in theoretical computer science is the deterministic finite automaton (DFA). Such an automaton is defined over a finite alphabet $\Sigma$ (for example $\Sigma = \{0, 1\}$). Then it has a finite set of states $Q$. The main ingredient is the transition function

$$\delta : Q \times \Sigma \to Q.$$ 

We can visualize this function as follows: whenever $\delta(q, \sigma) = q'$, we draw an arrow from state $q$ to state $q'$, labeled with $\sigma$.

Finally, there is a starting state $s \in Q$ and a subset $F \subseteq Q$ of accepting states. Figure 19 depicts a DFA with state set $Q = \{0, 1, 2\}$. The starting state is indicated by an arrow coming in from nowhere, and the accepting states are marked with double circles (in this case, there is only one).

![Figure 19: A deterministic finite automaton (DFA)](image)

Why can we call such an automaton a computer model? Because it performs a computation, namely the following: given an input word $w \in \Sigma^*$ (finite sequence of symbols from the alphabet $\Sigma$), the automaton either accepts, or rejects it. To do this, the word $w$ is processed symbol by symbol, starting in $s$. Whenever the automaton is in some state $q$ and the next symbol is $\sigma$, the automaton switches to state $q' = \delta(q, \sigma)$. When all symbols have been processed, the automaton is either in an accepting state $q \in F$ (in which case $w$ is accepted), or in a non-accepting state $q \in Q \setminus F$ (in which case $w$ is rejected).

For example, when we feed the automaton of Figure 19 with the word $w = 0101$, the sequence of states that are being visited is $0, 0, 1, 2, 2$. Consequently, $w$ is rejected.

The language $L$ of the automaton is the set of accepted words. This is a (generally infinite) subset of $\Sigma^*$. Let’s try to determine the language of the automaton in Figure 19.
It turns out that this is not such a straightforward task, and you need the right idea. (To be honest, we had the idea first and then came up with an automaton that realizes it). We claim that the automaton accepts exactly all the words that are divisible by 3 if you interpret the word as a binary number (where the empty word is interpreted as 0). For example, 0101 is the binary number

\[0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5,\]

and indeed 5 is not divisible by 3 (and hence rejected). In fact (and this is the key to the proof of our claim), the state after processing \(w\) is the one numbered with \(w \mod 3\). You can therefore say that the DFA of Figure 19 is a computer (with a built-in program) that can solve the decision problem of checking whether a given number is divisible by 3.

We are slowly approaching the actual challenge. For every subset \(L\) of \((0,1)^*\) from the following list, either find a DFA that has \(L\) as its language, or prove that such a DFA cannot exist (which would show that DFA are limited in their computational power).

a) \(L = \{w \in (0,1)^* \mid w\ has\ an\ even\ number\ of\ zeros\ and\ an\ even\ number\ of\ ones\}\)

b) \(L = \{w \in (0,1)^* \mid w\ is\ divisible\ by\ 5\ when\ interpreted\ as\ a\ binary\ number\}\)

c) \(L = \{w \in (0,1)^* \mid w\ has\ more\ zeros\ than\ ones\}\)

d) \(L = \{w \in (0,1)^* \mid w\ does\ not\ contain\ three\ consecutive\ ones\}\)

Exercise 110 A Sudoku puzzle is posed on a grid of 9×9 cells, subdivided into 9 square boxes of 3×3 cells each. Some grid cells are already filled by numbers between 1 and 9; the goal is to fill the remaining cells by numbers between 1 and 9 in such a way that within each row, column, and box of the completed grid, every number occurs exactly once. Here is an example of a Sudoku puzzle:

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

4 8
2
9 1
5

7 3
6
4

In solving the puzzle, one may try to deduce from the already filled numbers that exactly one number is a candidate for a suitable empty cell. Then this number is
filled into the cell, and the deduction process is repeated. There are two situations where such a
deduction for the cell in row r / column c and number n is particularly easy and follows the Sherlock Holmes approach ("How often have I said to you that
when you have eliminated the impossible, whatever remains, however improbable, must be the truth?").

1. All numbers distinct from n already appear somewhere in the same row, column, or 3x3 box. This necessarily means that the cell has to be filled with n, since we have eliminated all other numbers as impossible.

2. All other cells in the same row, or in the same column, or in the same 3x3 box are already known not to contain n. Again, the cell has to be filled by n then, since we have eliminated all other cells for the number n within the row, column, or box.

Write a program sudoku.cpp that takes as input a Sudoku puzzle in form of a
sequence of 81 numbers between 0 and 9 (the grid numbers given row by row, where
0 indicates an empty cell). The numbers might be separated by whitespaces, so that
the Sudoku puzzle from above could conveniently be encoded like this in a file:

0 0 0 1 0 0 7 4 0
0 5 0 0 9 0 0 3 2
0 0 6 7 0 0 9 0 0

4 0 0 8 0 0 0 0 0
0 2 0 0 0 0 0 1 0
0 0 0 0 9 0 0 0 5

0 0 4 0 0 7 3 0 0
7 3 0 0 0 0 6 0
0 6 5 0 0 4 0 0 0

The program should now try to solve the puzzle by using only the two Sherlock-
Holmes-type deductions from above. The output should be a (partially) completed
grid that is either the solution to the puzzle, or the unique (why?) partial solution
in which no Sherlock-Holmes-type deductions apply anymore (again, empty cells
should be indicated by the digit 0).

In the above example, the output of a correct program will be the solution:

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

4 7 9 8 5 1 6 2 3
5 2 8 6 7 3 4 1 9
6 1 3 2 4 9 8 7 5
For reading the input from a file, it can be convenient to redirect the standard input to the file containing the puzzle data. For checking whether any Sherlock-Holmes-type deductions apply, it can be useful to maintain (and update) for every triple \((r, c, n)\) the information whether \(n\) is still a possible candidate for the cell in row \(r\) / column \(c\).

You will discover that many Sudoku puzzles that typically appear in newspapers can be solved by your program and are therefore easy, even if they are labeled as medium or hard.

**Hint:** It is advisable not to optimize for efficiency here, since this will only lead to more complicated and error-prone code. Given the very small problem size, such optimizations won't have a noticeable effect anyway.

**Exercise 111** The two logicians Mr. Sum and Mr. Product are friends who one day have a phone conversation. Before, Mr. Sum only knows the sum \(s = a + b\) of two unknown integer numbers \(1 < a < 100\) and \(1 < b < 100\). Mr. Product, on the other hand, only knows the product \(p = a \cdot b\) of \(a\) and \(b\). The conversation goes as follows:

- Mr. Product: “I don't know the numbers \(a\) and \(b\).”
- Mr. Sum: “I knew that you don't know them.”
- Mr. Product: “Ah... but now I know them.”
- Mr. Sum: “Then I know them too, now.”

What is the set of numbers \(\{a, b\}\)? You have to assume, of course, that all statements made during this conversation are true. Write a program that computes \(\{a, b\}\), or submit a convincing written solution!
3.2 Recursion

Mir san mir.

Bavarian dictum, meaning "we are we".

Beware of bugs in the above code; I have only proved it correct, not tried it.

D./ E./ Knuth, in a letter to van Emde Boas (1977)

This section introduces recursive functions, functions that directly or indirectly call themselves. You will see that recursive functions are very natural in many situations, and that they lead to compact and readable code close to mathematical function definitions. We will also explain how recursive function calls are processed, and how recursion can (in principle) be replaced with iteration. In the end, you will see two applications (sorting, and drawing fractals) that demonstrate the power of recursion.

3.2.1 A warm-up

Many mathematical functions are naturally defined recursively, meaning that the function to be defined appears in its own definition. For example, for any $n \in \mathbb{N}$, the number $n!$ can recursively be defined as follows.

$$n! := \begin{cases} 
1, & \text{if } n \leq 1 \\
 n \cdot (n - 1)!, & \text{if } n > 1.
\end{cases}$$

In C++ we can also have recursive functions: a function may call itself. This is nothing exotic, since after all, a function call is just an expression that can in principle appear anywhere in the function's scope, and that scope includes the function body. Here is a recursive function for computing $n!$; in fact, this definition exactly matches the mathematical definition from above.

```cpp
// POST: return value is n!
unsigned int fac(const unsigned int n) {
    if (n <= 1) return 1;
    return n * fac(n-1); // n > 1
}
```

Here, the expression `fac(n-1)` is a recursive call of `fac`.
Infinite recursion. With recursive functions, we have the same issue as with loops (Section 2.4.2): it is easy to write down function calls whose evaluation does not terminate. Here is the shortest way of creating an infinite recursion: define the function

```c
void f()
{
    f();
}
```

with no arguments and evaluate the expression \( f() \). The reason for non-termination is clear: the evaluation of \( f() \) consists of an evaluation of \( f() \) which consists of an evaluation of \( f() \) which...you get the picture.

As for loops, the function definition has to make sure that progress towards termination is made in every function call. For the function \( \text{fac} \) above, this is the case: each time \( \text{fac} \) is called recursively, the value of the call argument becomes smaller, and when the value reaches 1, no more recursive calls are performed: we say that the recursion "bottoms out".

### 3.2.2 The call stack

Let's try to understand what exactly happens during the evaluation of \( \text{fac}(3) \), say. The formal argument \( n \) is initialized with 3, and since this is greater than 1, the statement \( \text{return } n * \text{fac}(n-1); \) is executed next. This first evaluates the expression \( n * \text{fac}(n-1) \) and in particular the right operand \( \text{fac}(n-1) \). Since \( n-1 \) has value 2, the formal argument \( n \) is therefore initialized with 2.

But wait: what is "the" formal argument? Automatic storage duration implies that each function call has its "own" fresh instance of the formal argument, and the lifetime of this instance is the respective function call. In evaluating \( f(n-1) \), we therefore get a new instance of the formal argument \( n \), on top of the previous instance from the call \( f(3) \) (that has not yet terminated). But which instance of \( n \) do we use in the evaluation of \( f(n-1) \)? Quite naturally, it will be the new one, the one that "belongs" to the call \( f(n-1) \). This rule is in line with the general scope rules from Section 2.4.3: the relevant declaration is always the most recent one that is still visible.

The technical realization of this is very simple. Everytime a function is called, the call argument is evaluated, and the resulting value is put on the call stack which is simply a region in the computer's memory. If the function has several arguments, several values are put on the call stack; to keep the description simple, we concentrate on the case of one argument.

Like a stack of papers on your desk, the call stack has the property that the object that came last is "on top". Upon termination of a function call, the top object is taken off the stack again. Whenever a function call accesses or changes its formal argument, it does so by accessing or changing the corresponding object on top of the stack.

This has all the properties we want: every function call works with its own instance of the formal argument; when it calls another function (or the function itself recursively),
this instance becomes temporarily hidden, until the nested call has terminated. At that point, the instance reappears on top of the stack and allows the original function call to work with it again.

Table 5 shows how this looks like for \( f(3) \), assuming that the right operand of the multiplication operator is always evaluated first. Putting an object on the stack “pushes” it, and taking the top object off “pops” it.

<table>
<thead>
<tr>
<th>call stack (bottom ( \leftarrow \rightarrow ) top)</th>
<th>evaluation sequence</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \cdots )</td>
<td>fac(3)</td>
<td>push 3</td>
</tr>
<tr>
<td>( \cdots ) n: 3</td>
<td>n * fac(n-1)</td>
<td></td>
</tr>
<tr>
<td>( \cdots ) n: 3</td>
<td>n * fac(2)</td>
<td>push 2</td>
</tr>
<tr>
<td>( \cdots ) n: 3 n: 2</td>
<td>n * (n * fac(n-1))</td>
<td></td>
</tr>
<tr>
<td>( \cdots ) n: 3 n: 2</td>
<td>n * (n * fac(1))</td>
<td>push 1</td>
</tr>
<tr>
<td>( \cdots ) n: 3 n: 2 n: 1</td>
<td>n * (n * 1)</td>
<td>pop</td>
</tr>
<tr>
<td>( \cdots ) n: 3 n: 2</td>
<td>n * (2 * 1)</td>
<td></td>
</tr>
<tr>
<td>( \cdots ) n: 3</td>
<td>3 * 2</td>
<td>pop</td>
</tr>
<tr>
<td>( \cdots ) n: 3</td>
<td>6</td>
<td>pop</td>
</tr>
</tbody>
</table>

Table 5: The call stack, and how it evolves during an evaluation of fac(3); the respective value of \( n \) to use is always the one on top

Because of the call stack, infinite recursions do not only consume time but also memory. Unlike infinite loops, they usually lead to a program abortion as soon as the memory reserved for the call stack is full.

3.2.3 Basic practice

Let us consider two more simple recursive functions that are somewhat more interesting than \( fac \). They show that recursive functions are particularly amenable to correctness proofs of their postconditions, and this makes them attractive. On the other hand, we also see that it is easy to write innocent-looking recursive functions that are very inefficient to evaluate.

Greatest common divisor. Consider the problem of finding the greatest common divisor \( \text{gcd}(a, b) \) of two natural numbers \( a, b \). This is defined as the largest natural number that divides both \( a \) and \( b \) without remainder. In particular, \( \text{gcd}(n, 0) = \text{gcd}(0, n) = n \) for \( n > 0 \); let us also define \( \text{gcd}(0, 0) := 0 \).

The Euclidean algorithm finds \( \text{gcd}(a, b) \), based on the following
Lemma 1 If $b > 0$, then
\[ \gcd(a, b) = \gcd(b, a \mod b). \]

Proof. Let $k$ be a divisor of $b$. From
\[ a = (a \div b)b + a \mod b \]
it follows that
\[ \frac{a}{k} = (a \div b) \frac{b}{k} + \frac{a \mod b}{k}. \]
Since $a \div b$ and $b/k$ are integers, we get
\[ \frac{a \mod b}{k} \in \mathbb{N} \iff \frac{a}{k} \in \mathbb{N}. \]
In words, if $k$ is a divisor of $b$, then $k$ divides $a$ if and only if $k$ divides $a \mod b$. This means, the divisors of $a$ and $b$ are exactly the divisors of $b$ and $a \mod b$. This proves that $\gcd(a, b)$ and $\gcd(b, a \mod b)$ are equal. \[\square\]

Here is the corresponding C++ function for computing the greatest common divisor of two unsigned int values, according to the Euclidean algorithm.

```
// POST: return value is the greatest common divisor of a and b
unsigned int gcd (const unsigned int a, const unsigned int b)
{
    if (b == 0) return a;
    return gcd(b, a % b); // b != 0
}
```

The Euclidean algorithm is very fast. We can easily call it for any unsigned int values on our platform, without noticing any delay in the evaluation.

Correctness and termination. For recursive functions, it is often very easy to prove that the postcondition is correct, by using the underlying mathematical definition directly (such as $n!$ for fac), or by using some facts that follow from the mathematical definition (such as Lemma 1 for gcd).

The correctness proof must involve a termination proof, so let's start with this: every call to gcd terminates, since the value $b$ of the second argument is bounded from below by 0 and gets smaller in every recursive call (we have $a \mod b < b$).

Given this, the correctness of the postcondition follows from Lemma 1 by induction on $b$. For $b = 0$, this is clear. For $b > 0$, we inductively assume that the postcondition is correct for all calls to gcd where the second argument has value $b' < b$. Since $b' = a \mod b$ satisfies $b' < b$, we may assume that the call $\gcd(b', a \ % b)$ correctly returns $\gcd(b, a \mod b)$. But by the lemma, $\gcd(b, a \mod b) = \gcd(a, b)$, so the statement
\[
\text{return gcd}(b, a \ % b);
\]
correctly returns $\gcd(a, b)$. 

Fibonacci numbers. The sequence \(0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots\) of Fibonacci numbers is one of the most famous sequences in mathematics. Formally, the sequence is defined as follows.

\[
F_0 := 0, \\
F_1 := 1, \\
F_n := F_{n-1} + F_{n-2}, \quad n > 1.
\]

This means, every element of the sequence is the sum of the two previous ones. From this definition, we can immediately write down a recursive C++ function for computing Fibonacci numbers, getting termination and correctness for free.

```cpp
// POST: return value is the n-th Fibonacci number F_n
unsigned int fib (const unsigned int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib (n -1) + fib (n -2); // n > 1
}
```

If you write a program to compute the Fibonacci number \(F_n\) using this function, you will notice that somewhere between \(n = 30\) and \(n = 50\), the program becomes very slow. You even notice how much slower it becomes when you increase \(n\) by just 1.

The reason is that the mathematical definition of \(F_n\) does not lead to an efficient algorithm, since all values \(F_i, i < n - 1\), are repeatedly computed, some of them extremely often. You can for example check that the call to \(\text{fib}(50)\) computes \(F_{48}\) already twice (once directly in \(\text{fib}(n-2)\), and once indirectly from \(\text{fib}(n-1)\). \(F_{47}\) is computed three times, \(F_{46}\) five times, and \(F_{45}\) eight times (do you see a pattern?).

### 3.2.4 Recursion versus iteration

From a strictly functional point of view, recursion is superfluous, since it can be simulated through iteration (and a call stack explicitly maintained by the program; we could simulate the call stack with an array). We don't have the means to prove this here, but we want to show it for the recursive functions that we have seen in the previous section.

The function \(\gcd\) is very easy to write iteratively, since it is tail-end recursive. This means that there is only one recursive call, and that one appears at the very end of the function body. Tail-end recursion can be replaced by a simple loop that iteratively updates the formal arguments until the termination condition is satisfied. In the case of \(\gcd\), this update corresponds to the transformation \((a, b) \rightarrow (b, a \mod b)\).

```cpp
// POST: return value is the greatest common divisor of a and b
unsigned int gcd2 (unsigned int a, unsigned int b) {
    while (b != 0) {
        const unsigned int a_prev = a;
        a = b;
        b = a_prev % b;
    }
    return a;
}
```
a = b;
    b = a_prev % b;
}
    return a;

You see that we get longer and less readable code, and that we need an extra variable to remember the previous value of \( a \) before the update step; in the spirit of Section 2.4.8 we should therefore use the original recursive formulation.

Our function \( \text{fib} \) for computing Fibonacci numbers is not tail-end recursive, but it is still easy to write it iteratively. Remember that \( F_n \) is the sum of \( F_{n-1} \) and \( F_{n-2} \). We can therefore write a loop whose iteration \( i \) computes \( F_i \) from the previously computed values \( F_{i-2} \) and \( F_{i-1} \) that we maintain in the variables \( a \) and \( b \).

// POST: return value is the n-th Fibonacci number \( F_n \)
unsigned int \text{fib2} ( \text{const unsigned int n})
{
    if (n == 0) return 0;
    if (n <= 2) return 1;
    unsigned int a = 1; // \( F_1 \)
    unsigned int b = 1; // \( F_2 \)
    for (unsigned int i = 3; i <= n; ++i) {
        const unsigned int a_prev = a; // \( F_{i-2} \)
        a = b; // \( F_{i-1} \)
        b += a_prev; // \( F_{i-1} + F_{i-2} \rightarrow F_i \)
    }
    return b;
}

Again, this non-recursive version \( \text{fib2} \) is substantially longer and more difficult to understand than \( \text{fib} \), but this time there is a benefit: \( \text{fib2} \) is much faster, since it computes every number \( F_i, i \leq n \) exactly once. While we would grow old in waiting for the call \( \text{fib}(50) \) to terminate, \( \text{fib2}(50) \) gives us the answer in no time. (Unfortunately, this answer may be incorrect, since \( F_{50} \) could exceed the value range of the type unsigned int.)

In this case we would prefer \( \text{fib2} \) over \( \text{fib} \), simply since \( \text{fib} \) is too inefficient for practical use. The more complicated function definition of \( \text{fib2} \) is a moderate price to pay for the speedup that we get.

### 3.2.5 Primitive recursion

Roughly speaking, a mathematical function is \textit{primitive recursive} if it can be written as a C++ function \( f \) in such a way that \( f \) neither directly nor indirectly calls itself with call arguments depending on \( f \). For example,

\[
\text{unsigned int } f ( \text{const unsigned int n})
\]
is not allowed, since $f$ recursively calls itself with an argument depending of $f$. This does not mean that the underlying mathematical function is not primitive recursive, it just means that we have chosen the wrong C++ implementation. Indeed, the above $f$ implements the mathematical function satisfying $f(n) = 1$ for all $n$, and this function is obviously primitive recursive.

In the early 20-th century, it was believed that the functions whose values can in principle be computed by a machine are exactly the primitive recursive ones. Indeed, the function values one computes in practice (including $\text{gcd}(a, b)$ and $F_n$) come from primitive recursive functions.

It later turned out that there are computable functions that are not primitive recursive. A simple and well-known example is the binary Ackermann function $A(m, n)$, defined by

$$A(m, n) = \begin{cases} n + 1, & \text{if } m = 0 \\ A(m - 1, 1), & \text{if } m > 0, n = 0 \\ A(m - 1, A(m, n - 1)), & \text{if } m > 0, n > 0. \end{cases}$$

The fact that this function is not primitive recursive requires a proof (that we don’t give here). As already noted above, it is necessary but not sufficient that this definition recursively uses $A$ with an argument that depends on $A$.

It may not be immediately clear that the corresponding C++ function

```cpp
// POST: return value is the Ackermann function value A(m,n)
unsigned int A ( const unsigned int m, const unsigned int n) {
    if (m == 0) return n+1;
    if (n == 0) return A(m-1,1);
    return A(m-1, A(m, n-1));
}
```

always terminates, but Exercise 113 asks you to show this. Table 6 lists some Ackermann function values. For $m \leq 3$, $A(m, n)$ looks quite moderate, but starting from $m = 4$, the values get extremely large. You can still compute $A(4, 1)$, although this takes surprisingly long already. You might be able to compute $A(4, 2)$; after all, $2^{65536} - 3$ has “only” around 20,000 decimal digits. But the call to $A(4, 3)$ will not terminate within any observable period.

It can in fact be shown that $A(n, n)$ grows faster than any primitive recursive function in $n$ (and this is a proof that $A$ cannot be primitive recursive). Recursion is a powerful but also dangerous tool, since it is easy to encode (too) complicated computations with very few lines of code.
### 3.2. Sorting

Sorting a sequence of values (numbers, texts, etc.) into ascending order is a very basic and important operation. For example, a specific value can be found much faster in a sorted than in an unsorted sequence (see Exercise 121). You know this from daily life, and that’s why you sort your CDs, and why the entries in a telephone directory are sorted by name.

We have asked you in Exercise 83 to write a program that sorts a given sequence of integers; Exercise 102 was about making this into a function that sorts all numbers described by a given pointer range. In both exercises, you were not supposed to do any efficiency considerations.

Here we want to catch up on this and investigate the complexity of the sorting problem. Roughly speaking, the complexity of a problem is defined as the complexity (runtime) of the fastest algorithm that solves the problem. In computing Fibonacci numbers in Section 3.2.3 and Section 3.2.4, we have already seen that the runtimes of different algorithms for the same problem may vary a lot. The same is true for sorting algorithms, as we will discover shortly.

Let us start by analyzing one of the “obvious” sorting algorithms that you may have come up with in Exercise 83. The simplest one that the authors can think of is minimum-sort. Given the sequence of values (let’s assume they are integers), minimum-sort first finds the smallest element of the sequence; then it interchanges this element with the first element. The sequence now starts with the smallest element, as desired, but the remainder of the sequence still needs to be sorted. But this is done in the same way: the smallest element among the remaining ones is found and interchanged with the second element of the sequence, and so on.

Assuming that the sequence is described by a pointer range \([\text{first}, \text{last})\), minimum-sort can be realized as follows.

```c
// PRE: \([\text{first}, \text{last})\) is a valid range
// POST: the elements *p, p in \([\text{first}, \text{last})\) are in ascending order
void minimum_sort (int* first, int* last)
{
```

<table>
<thead>
<tr>
<th>(n)</th>
<th>(n+1)</th>
<th>(n+2)</th>
<th>(2n+3)</th>
<th>(2n+5-3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>2 3</td>
<td>4 5</td>
<td>(2^{n+3}-3)</td>
<td></td>
</tr>
<tr>
<td>2 3</td>
<td>5 7</td>
<td>9</td>
<td>(2^{n+5-3})</td>
<td></td>
</tr>
<tr>
<td>3 5</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 13</td>
<td>(2^{65536}-3)</td>
<td>(2^{65536}-3)</td>
<td>(2^{2-\frac{3}{n+3}}-3)</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Some values of Ackermann's function
 CHAPTER 3. FUNCTIONS

```cpp
for (int* p = first; p != last; ++p) {
    // find minimum in nonempty range described by [p, last)
    int* p_min = p;      // pointer to current minimum
    int* q = p;          // pointer to current element
    while (++q != last)  
        if (*q < *p_min) p_min = q;
    // interchange *p with *p_min
    std::iter_swap(p, p_min);
}
```

The standard library function `std::iter_swap` interchanges the values of the objects pointed to by its two arguments. There is also a function `std::min_element` (with the same functionality as our home-made `min_element` on page 190) that we could use to get rid of the inner loop; however, since we want to analyze the function `minimum_sort` in detail, we refrain from calling any nontrivial standard library function here.

What can we say about the runtime of `minimum_sort` for a given range? That it depends on the platform, this is for sure. On a modern PC, the algorithm will run much faster than on a vintage computer from the twentieth century. There is no such thing as “the” runtime. But if we look at what the algorithm does, we can find a measure of runtime that is platform-independent.

A dominating operation in the sense that it occurs very frequently during a call to `minimum_sort` is the comparison \(*q < *p\_min\). We can even exactly count the number of such comparisons, depending on the number of elements \(n\) that are to be sorted. In the first execution of the `while` statement, the first element is compared with all \(n - 1\) succeeding elements. In the second execution, the second element in compared with all the \(n - 2\) succeeding elements, and so on. In the second-to-last execution of the `while` statement, finally, we have one comparison, and that’s it. We therefore have the following

**Observation 1** The function `minimum_sort` sorts a sequence of \(n\) elements with

\[
1 + 2 + \ldots + n - 1 = \frac{n(n - 1)}{2}
\]

comparisons between sequence elements.

Why do we specifically count these comparisons? Because any other operation is either performed much less frequently (for example, the declaration statement `int* q = p` is executed only \(n\) times), or with approximately the same frequency. This concerns the assignment `p\_min = q` which may happen up to \(n(n - 1)/2\) times, and the expression `++q != last`; this one is evaluated even more frequently, namely \(n(n - 1)/2 + n\) times. The total number of operations is therefore at most \(c_1 n(n - 1)/2 + c_2 n\) for some constants \(c_1, c_2\). For large \(n\), the linear term \(c_2 n\) is negligible compared to the quadratic term \(c_1 n(n - 1)/2\); we can therefore conclude that the total number of operations needed to sort \(n\) numbers is proportional to the number of comparisons between sequence elements.
This implies the following: if you measure the runtime of the whole sorting algorithm, the resulting time $T_{\text{total}}$ will be proportional to the time $T_{\text{comp}}$ that is being spent with comparisons between sequence elements. (However, due to the effects of caching and other add-ons to the von-Neumann architecture, this is not necessarily true on every platform.) Since $T_{\text{comp}}$ is in turn proportional to the number of comparisons itself, this number is a good indicator for the efficiency of the algorithm.

If you think about sorting more complicated values (like names in a telephone directory), a comparison between two elements might even become the single most time-consuming operation. In such a scenario, $T_{\text{comp}}$ may eat up almost everything of $T_{\text{total}}$, making the comparison count an even more appropriate measure of efficiency.

To check that all this is not only pure theory, let us make some experiments and measure the time that it takes to execute Program 28 below, for various values of $n$ (read from a file in order not to measure the time it takes us to enter $n$). The program first brings the sequence $0, 1, \ldots, n-1$ into random order, using the standard library function `std::random_shuffle`. Then it calls the function `minimum_sort` and finally checks whether we now indeed have the ascending sequence $0, 1, \ldots, n-1$. Yes, this program does other things apart from the actual sorting, but all additional operations are "cheap" in the sense that their number is proportional to $n$ at most; according to our above line of arguments, they should therefore not matter.

```cpp
1 // Prog: minimum_sort.cpp
2 // implements and tests minimum-sort on random input
3
4 #include <iostream>
5 #include <algorithm>
6
7 // PRE: [first, last) is a valid range
8 // POST: the elements *p, p in [first, last) are in ascending order
9 void minimum_sort (int * first, int * last)
10 {
11    for (int * p = first; p != last; ++p) {
12        // find minimum in nonempty range described by [p, last)
13        int * p_min = p; // pointer to current minimum
14        int * q = p;    // pointer to current element
15        while (++q != last)
16            if (*q < *p_min) p_min = q;
17        // interchange *p with *p_min
18        std::iter_swap (p, p_min);
19    }
20 }
21
22 int main()
23 {
24    // input of number of values to be sorted
```
Table 7: Number of comparisons and runtime of minimum-sort

Table 7 summarizes the results. For every value of \( n \), \( G_{\text{comp}} \) is the number of Gigacomparisons \((10^9 \text{ comparisons})\), according to Observation 1. In other words, \( G_{\text{comp}} = 10^9 n(n - 1)/2 \). Time is the absolute runtime of the program in minutes and seconds, on a modern PC. \( \text{sec/Gcomp} \) is Time (in seconds) divided by \( G_{\text{comp}} \) and tells us how many seconds the program needs to perform one Gigacomparison.

The table shows that the number of seconds per Gigacomparison is around 3 for all considered values of \( n \). As predicted above, the runtime in practice is therefore indeed proportional to the number of comparisons between sequence elements. This number quadruples from one column to the next, and so does the runtime.

We also see that sorting numbers using minimum-sort appears to be pretty inefficient. 1,600,000 is not large by today’s standards, but to sort that many numbers takes more than one hour! Given that \( \text{sec/Gcomp} \) appears to be constant, we can even estimate the time that it would take to sort 10,000,000 numbers, say. For this, we derive from
Observation 3.2: the required number of Gigacomparisons (50,000) and multiply it with 3. The resulting 150,000 seconds are almost two days.

Essentially the same figures result from running other well-known simple sorting algorithms like bubble-sort or insert-sort. Can we do better? Yes, we can, and recursion helps us to do it!

Merge-sort. The paradigm behind the merge-sort algorithm is this: if a problem is too large to be solved directly, subdivide it into smaller subproblems that are easier to solve, and then put the overall solution together from the solutions of the subproblems. This paradigm is known as divide and conquer.

Here is how this works for sorting. Let us imagine that the numbers to be sorted come as a deck of cards, with the numbers written on them. The first step is to partition the deck into two smaller decks of half the size each. These two decks are then sorted independently from each other, with the same method; but the main ingredient of this method comes only now: we have to merge the two sorted decks into one sorted deck. But this is not hard: we put the two decks in front of us (both now have their smallest card on top); as long as there are still cards in one or both of the decks, the smaller of the two top cards (or the single remaining top card) is taken off and put upside down on a new deck that in the end represents the result of the overall sorting process. Figure 20 visualizes the merge step.

Here is how merge-sort can be realized in C++, assuming that we have a function merge that performs the above operation of merging two sorted sequences into one sorted sequence.

```cpp
// PRE: [first, last) is a valid range
// POST: the elements *p, p in [first, last) are in ascending order
void merge_sort (int* first, int* last)
{
    const int n = last - first;
    if (n <= 1) return;  // nothing to do
    int* const middle = first + n/2;
    merge_sort (first, middle);  // sort first half
    merge (first, middle, last);
}
```

Figure 20: Merging two sorted decks of cards into one sorted deck
merge_sort (middle, last); // sort second half
merge (first, middle, last); // merge both halves
}

If there is more than one element to sort, the function splits the range [first, last) into two ranges [first, middle) and [middle, last) of lengths \([n/2]\) and \([n/2]\). Just as a reminder, for any real number \(x\), \([x]\) is the smallest integer greater or equal to \(x\) ("\(x\) rounded up"), and \([x]\) is the largest integer smaller or equal to \(x\) ("\(x\) rounded down"). If \(n\) is even, both values \([n/2]\) and \([n/2]\) are equal to \(n/2\), and otherwise, the first value is smaller by one.

As its next step, the algorithm recursively sorts the elements described by both ranges. In the end, it calls the function merge on the two ranges. In commenting the latter function, we stick to the deck analogy that we have used above. If you have understood the deck merging process, you will perceive the definition of merge as being straightforward, despite the pointer handling.

// PRE: [first, middle), [middle, last) are valid ranges; in // both of them, the elements are in ascending order void merge (int* first, int* middle, int* last)
{
    const int n = last - first; // total number of cards
    int* const deck = new int[n]; // new deck to be built

    int* left = first; // top card of left deck
    int* right = middle; // top card of right deck
    for (int* d = deck; d != deck + n; ++d)
        // put next card onto new deck
        if (left == middle) *d = *right ++; // left deck is empty
        else if (right == last) *d = *left ++; // right deck is empty
        else if (*left < *right) *d = *left ++; // smaller top card left
        else *d = *right ++; // smaller top card right

    // copy new deck back into [first, last)
    const int* d = deck;
    while (first != middle) *first ++ = *d ++;
    while (middle != last) *middle ++ = *d ++;

    delete [] deck;
}

Analyzing merge-sort. As for minimum-sort, we will count the number of comparisons between sequence elements that occur when a sequence of \(n\) numbers is being sorted. Again, we can argue that the total number of operations is proportional to this number of comparisons. For merge-sort, this fact is not so immediate, though, and we don't
expect you to understand it now. But for the benefit of (not only) the skeptic reader, we will check this fact experimentally below, as we did for minimum-sort.

All the comparisons take place during the calls to the function merge at the various levels of recursion, so let us first count the number of comparisons between sequence elements that one call to merge performs in order to create a sorted deck of \( n \) cards from two sorted decks.

It is apparent from the function body (and also from our informal description of the merging process above) that at most one comparison is needed for every card that is put on the new deck. Indeed, we may have to compare the two top cards of the left and the right deck in order to find out which card to take off next. But if one of the two decks becomes empty (this situation definitely occurs before the last card is put on the new deck), we don't do any further comparisons. This means that at most \( n - 1 \) comparisons between sequence elements are performed in merging two sorted decks into one sorted deck with \( n \) cards.

Knowing this, we can now prove our main result.

**Theorem 2** The function merge_sort sorts a sequence of \( n \geq 1 \) elements with at most

\[
(n - 1)[\log_2 n]
\]

comparisons between sequence elements.

**Proof.** We define \( T(n) \) to be the maximum possible number of comparisons between sequence elements that can occur during a call to merge_sort with an argument range of length \( n \). For example, \( T(0) = T(1) = 0 \), since for ranges of lengths 0 and 1, no comparisons are made. We also get \( T(2) = 1 \), since for a range of length 2, merge-sort performs one comparison (in merging two sorted decks of one card each into one sorted deck of two cards). In a similar way, we can convince ourselves that \( T(3) = 2 \). There are sequences of length 3 for which one comparison suffices (the first card may be taken off the left deck which consists only of one card), but the maximum number that defines \( T(3) \) is 2.

For general \( n \geq 2 \), we have the following recurrence relation:

\[
T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + n - 1.
\]  

(3.1)

To see this, let us consider a sequence of \( n \) elements that actually requires the maximum number of \( T(n) \) comparisons. This number of comparisons is the sum of the respective numbers in sorting the left and the right half, plus the number of comparisons during the merge step. The former two numbers are (by construction of merge_sort and definition of \( T \)) at most \( T(\lfloor n/2 \rfloor) \) and \( T(\lceil n/2 \rceil) \), while the latter number is at most \( n - 1 \) by our previous considerations regarding merge. It follows that \( T(n) \), the actual number of comparisons, is bounded by the sum of all three numbers.

Now we can prove the actual statement of the theorem. Since the merge-sort algorithm is recursive, it is natural that the proof is inductive. For \( n = 1 \), we have \( T(1) = 0 = (1 - 1)[\log_2 2] \), so the statement holds for \( n = 1 \).
For $n \geq 2$, let us assume that the statement of the theorem holds for all values in $\{1, \ldots, n-1\}$ (this is the inductive hypothesis). From this hypothesis, we need to derive the validity of the statement for the number $n$ itself (note that $\lfloor n/2 \rfloor, \lceil n/2 \rceil \geq 1$). This goes as follows.

$$\begin{align*}
T(n) & \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n - 1 \quad \text{(Equation 3.1)} \\
& \leq (\lfloor n/2 \rfloor - 1)\lceil \log_2 \lfloor n/2 \rfloor \rceil + (\lceil n/2 \rceil - 1)\lfloor \log_2 \lceil n/2 \rceil \rfloor + n - 1 \quad \text{(inductive hypothesis)} \\
& \leq \left( \lfloor n/2 \rfloor - 1 \right)(\lceil \log_2 n \rceil - 1) + \left( \lceil n/2 \rceil - 1 \right)(\lfloor \log_2 n \rfloor - 1) + n - 1 \quad \text{(Exercise 122)} \\
& = (n - 2)(\lceil \log_2 n \rceil - 1) + n - 1 \quad \text{(Exercise 122)} \\
& \leq (n - 1)(\lceil \log_2 n \rceil - 1) + n - 1 \\
& = (n - 1)\lceil \log_2 n \rceil.
\end{align*}$$

\[\square\]

As for minimum-sort, let us conclude with some experiments to check whether the number of comparisons between sequence elements is indeed a good indicator for the runtime in practice. The results of running merge-sort (Program 29 below) are given in Table 8 and look very different from the ones in Table 7.

---

For $n \geq 2$, let us assume that the statement of the theorem holds for all values in $\{1, \ldots, n-1\}$ (this is the inductive hypothesis). From this hypothesis, we need to derive the validity of the statement for the number $n$ itself (note that $\lfloor n/2 \rfloor, \lceil n/2 \rceil \geq 1$). This goes as follows.

$$\begin{align*}
T(n) & \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n - 1 \quad \text{(Equation 3.1)} \\
& \leq (\lfloor n/2 \rfloor - 1)\lceil \log_2 \lfloor n/2 \rfloor \rceil + (\lceil n/2 \rceil - 1)\lfloor \log_2 \lceil n/2 \rceil \rfloor + n - 1 \quad \text{(inductive hypothesis)} \\
& \leq \left( \lfloor n/2 \rfloor - 1 \right)(\lceil \log_2 n \rceil - 1) + \left( \lceil n/2 \rceil - 1 \right)(\lfloor \log_2 n \rfloor - 1) + n - 1 \quad \text{(Exercise 122)} \\
& = (n - 2)(\lceil \log_2 n \rceil - 1) + n - 1 \quad \text{(Exercise 122)} \\
& \leq (n - 1)(\lceil \log_2 n \rceil - 1) + n - 1 \\
& = (n - 1)\lceil \log_2 n \rceil.
\end{align*}$$

\[\square\]

As for minimum-sort, let us conclude with some experiments to check whether the number of comparisons between sequence elements is indeed a good indicator for the runtime in practice. The results of running merge-sort (Program 29 below) are given in Table 8 and look very different from the ones in Table 7.

---

```cpp
1 // Prog: merge_sort.cpp
2 // implements and tests merge-sort on random input
3
4 #include <iostream>
5 #include <algorithm>
6
7 // PRE: [first, middle), [middle, last) are valid ranges; in
8 // both of them, the elements are in ascending order
9 void merge (int* first, int* middle, int* last)
10 {
11     const int n = last - first; // total number of cards
12     int* const deck = new int[n]; // new deck to be built
13
14     int* left = first; // top card of left deck
15     int* right = middle; // top card of right deck
16     for (int* d = deck; d != deck + n; ++d)
17         // put next card onto new deck
```

---

```cpp
1 // Prog: merge_sort.cpp
2 // implements and tests merge-sort on random input
3
4 #include <iostream>
5 #include <algorithm>
6
7 // PRE: [first, middle), [middle, last) are valid ranges; in
8 // both of them, the elements are in ascending order
9 void merge (int* first, int* middle, int* last)
10 {
11     const int n = last - first; // total number of cards
12     int* const deck = new int[n]; // new deck to be built
13
14     int* left = first; // top card of left deck
15     int* right = middle; // top card of right deck
16     for (int* d = deck; d != deck + n; ++d)
17         // put next card onto new deck
```
3.2. RECURSION

if (left == middle) *d = *right++; // left deck is empty
else if (right == last) *d = *left++; // right deck is empty
else if (*left < *right) *d = *left++; // smaller top card left
else *d = *right++; // smaller top card right

// copy new deck back into [first, last)
const int* d = deck;
while (first != middle) *first ++ = *d ++;
while (middle != last) *middle ++ = *d ++;
delete[] deck;

// PRE: [first, last) is a valid range
// POST: the elements *p, p in [first, last) are in ascending order
void merge_sort (int* first, int* last)
{
    const int n = last - first;
    if (n <= 1) return; // nothing to do
    int* const middle = first + n/2;
    merge_sort (first, middle); // sort first half
    merge_sort (middle, last); // sort second half
    merge (first, middle, last); // merge both halves
}

int main()
{
    // input of number of values to be sorted
    int n;
    std::cin >> n;
    int* const a = new int[n];
    std::cout << "Sorting " << n << " integers...\n";
    // create sequence:
    for (int i=0; i<n; ++i) a[i] = i;
    std::random_shuffle (a, a+n);
    // sort into ascending order
    merge_sort (a, a+n);
    // is it really sorted ?
    for (int i=0; i<n-1;++i)
        if (a[i] != i) std::cout << "Sorting error!\n";
Since \texttt{merge\_sort} incurs much less comparisons than \texttt{minimum\_sort}, our unit here is just \texttt{Mcomp}, the number of Megacomparisons (\(10^6\) comparisons), according to Theorem 2. In other words, \(\texttt{Mcomp} = 10^{-6}(n-1)\lceil\log_2 n\rceil\). \texttt{Time} is the absolute runtime of the program, this time in milliseconds and not minutes. But as in Table 7, \texttt{sec/Gcomp} tells us how many seconds the program needs to perform one Gigacomparison.

We first observe that this latter number is around 25 for all \(n\), also confirming in this case that the runtime is proportional to the number of comparisons. On the other hand, the time needed by \texttt{merge\_sort} per Gcomp is much higher than in \texttt{minimum\_sort} (25 seconds for \texttt{merge\_sort} vs. 3 seconds for \texttt{minimum\_sort}). It may be surprising that the difference is this large, but the fact that it is larger can be explained. \texttt{Merge\_sort} is a more complicated algorithm than \texttt{minimum\_sort}, with its recursive structure, the extra memory needed for the new deck, etc. The price to pay is that less comparisons can be done per second, since a lot of time is needed for other operations. But this is a moderate price, since we can more than pay for it by the gain in total runtime.

This gets us to the most positive news of Table 8: \texttt{Merge\_sort} is actually a very practical sorting algorithm. While it takes \texttt{minimum\_sort} more than one hour to process 1,600,000 numbers, \texttt{merge\_sort} does the same in less than a second. Our experimental results show that this is mainly due to the fact that \texttt{merge\_sort} needs much less comparisons (at most \((n-1)\lceil\log_2 n\rceil\)) than \texttt{minimum\_sort} with its \(n(n-1)/2\) comparisons. Still, \texttt{merge\_sort} is not the best sorting algorithm in practice, see Exercise 120.

### 3.2.7 Lindenmayer systems

In this final section we want to present another application in which recursion is predominant and difficult to avoid (an iterative version would indeed require an explicit stack). As a bonus, this applications lets us draw beautiful pictures.

Let us first fix an \textit{alphabet} \(\Sigma\) which is simply a finite set of symbols, for example \(\Sigma = \{F, +, -\}\). Let \(\Sigma^*\) denote the set of all \textit{words} that we can form from symbols in \(\Sigma\). For example, \(F + F + \in \Sigma^*\).

Next, we fix a function \(P : \Sigma \rightarrow \Sigma^*\). \(P\) maps every symbol to a word, and these are the \textit{productions}. We might for example have the productions

\[
\begin{align*}
\sigma & \; \rightarrow \; P(\sigma) \\
F & \; \rightarrow \; F + F + \\
+ & \; \rightarrow \; + \\
- & \; \rightarrow \; -
\end{align*}
\]
3.2. **RECURSION**

Finally, we fix an initial word \( s \in \Sigma^* \), for example \( s = F \).

The triple \( \mathcal{L} = (\Sigma, P, s) \) is called a *Lindenmayer system*. Such a system generates an infinite sequence of words \( s = w_0, w_1, \ldots \) as follows. To get the next word \( w_i \) from the previous word \( w_{i-1} \), we simply substitute all symbols in \( w_{i-1} \) by their productions.

In our example, this yields

\[
\begin{align*}
w_0 &= F, \\
w_1 &= F + F + \\
w_2 &= F + F + F + F + + \\
w_3 &= F + F + F + F + F + + F + F + F + + + \\
&\vdots
\end{align*}
\]

The next step is to “draw” these words, and this gives the pictures we were talking about.

**Turtle graphics.** Imagine a turtle sitting at some point \( p \) on a large piece of paper, with its head pointing in some direction, see Figure 21 (left). The turtle can understand the commands \( F, +, \) and \( - \). \( F \) means “move one step forward”, \( + \) means “turn counterclockwise by an angle of 90 degrees”, and \( - \) means “turn clockwise by an angle of 90 degrees”. The turtle can process any sequence of such commands, by executing them one after another. We are interested in the resulting path taken by the turtle on the piece of paper. The path generated by the command sequence \( F + F+ \), for example, is shown in Figure 21 (right), along with the position and orientation of the turtle after processing the command sequence.

The turtle can therefore graphically interpret any word generated by a Lindenmayer system over the alphabet \( \{F, +, -\} \).

**Recursively drawing Lindenmayer systems.** For \( \sigma \in \Sigma \), let \( w_i^\sigma \) denote the word resulting from \( \sigma \) by the \( i \)-fold substitution of all symbols according to their productions. In our running example, we have for example \( w_2 = w_2^F = F + F + + F + F + + \) and \( w_i^+ = + \) for all \( i \).

The point is now that we can express \( w_i^\sigma \) in terms of the \( w_{i-1} \)'s of other symbols, and this is where recursion comes into play. Suppose that \( P(\sigma) = \sigma_1 \cdots \sigma_k \). Then we can
obtain \( w^\sigma_i \) as follows. We first substitute \( \sigma \) by \( \sigma_1 \cdots \sigma_k \) (1-fold substitution), and in the resulting word \( \sigma_1 \cdots \sigma_k \) we apply \((i-1)\)-fold substitution to all the symbols. This shows that

\[
w^\sigma_i = w^\sigma_{i-1} \cdots w^\sigma_{i-1}.
\]

This formula also implies that the drawing of \( w^\sigma_i \) is obtained by simply concatenating the drawings for \( w^\sigma_{i-1}, \ldots, w^\sigma_{i-1} \). To get the actual word \( w_i \), we simply concatenate the drawings of all \( w^\sigma_i \) for \( \sigma \) running through the symbols of the initial word \( s \).

Program 30 shows how this works for our running example with productions \( F \rightarrow F + F +, + \rightarrow +, - \rightarrow - \) and initial word \( F \). Since \( P^i(+) = +, P^i(-) = - \) for all \( i \), we do not need to substitute \(+\) and \(-\) and get

\[
w_i = w^F_i = w^F_{i-1} + w^F_{i-1} + \quad (3.2)
\]

The program assumes the existence of a library turtle with predefined turtle command functions \texttt{forward}, \texttt{left} (counterclockwise rotation with some angle) and \texttt{right} (clockwise rotations with some angle) in namespace \texttt{ifm}.

In the documentation of the program, we have omitted the “trivial” productions \(+ \rightarrow +, - \rightarrow -\), and in specifying a Lindenmayer system, we can do so as well: we will usually only list productions for symbols that are not mapped to themselves.

```cpp
1 // Prog: lindenmayer.cpp
2 // Draw turtle graphics for the Lindenmayer system with
3 // production \( F \rightarrow F+F\) and initial word \( F \).
4
5 #include <iostream>
6 #include <IFM/turtle>
7
8 // POST: the word \( w_i^F \) is drawn
9 void f (const unsigned int i) {  
10    if (i == 0)  
11        ifm::forward();  // \( F \)
12    else {
13        f(i-1);  // \( w_{i-1}^F \)
14        ifm::left(90);  // +
15        f(i-1);  // \( w_{i-1}^F \)
16        ifm::left(90);  // +
17    }
18}
19
20 int main () {  
21    std::cout << "Number of iterations =? ";
22    unsigned int n;
23    std::cin >> n;
```
3.2. RECURSION

24  // draw \( w_n = w_n(F) \)
25  f(n);
26  return 0;
27 }

Program 30: ../progs/lecture/lindenmayer.cpp

For input \( n = 14 \), the program will produce the following drawing.

![Lindenmayer System Drawing]

As \( n \) gets larger, the picture does not seem to change much; it rotates, and some more details develop, but apart from that the impression is the same. Assume you could draw the picture for \( n = \infty \). Then equation (3.2) would give

\[
{w}_\infty = {w}_\infty + {w}_\infty + .
\]

This is a self-similarity: the drawing of \( w_\infty \) consists of two rotated drawings of itself. We have a fractal!

Additional features. We can extend the definition of a Lindenmayer system to include a rotation angle \( \alpha \) that may be different from 90 degrees. This is shown in Program 31 that draws a snowflake for input \( n = 5 \).
// Prog: snowflake.cpp
// Draw turtle graphics for the Lindenmayer system with
// production F -> F-F++F-F, initial word F++F++F and
// rotation angle 60 degrees.
#include <iostream>
#include <IFM/turtle>

// POST: the word $w_n^F$ is drawn
void f (const unsigned int i) {
    if (i == 0)
        ifm::forward(); // F
    else {
        f(i-1); // $w_{i-1}^F$
        ifm::right(60); // -
        f(i-1); // $w_{i-1}^F$
        ifm::left(120); // ++
        f(i-1); // $w_{i-1}^F$
        ifm::right(60); // -
        f(i-1); // $w_{i-1}^F$
    }
}

int main () {
    std::cout << "Number of iterations =? ";
    unsigned int n;
    std::cin >> n;
    // draw $w_n = w_n^F++w_n^F++w_n^F$
    f(n); // $w_n^F$
3.2. RECURRENCE

30 ifm::left(120); // ++
31 f(n); // w_n ^F
32 ifm::left(120); // ++
33 f(n); // w_n ^F
34 }
35 return 0;
36 }

Program 31: ../progs/lecture/snowflake.cpp

To get more flexibility, we can also extend the alphabet Σ of symbols. For example, we may add symbols without any graphical interpretation; these are still useful, though, since they may be used in productions. For example, the Lindenmayer system with Σ = {F,+,−,X,Y}, initial word X and productions

\[
\begin{align*}
X & \rightarrow X + YF + \\
Y & \rightarrow -FX -Y
\end{align*}
\]

yields the dragon curve (w_{14}, angle of 90 degrees).

The corresponding code is shown in Program 32.

1 // Prog: dragon.cpp
2 // Draw turtle graphics for the Lindenmayer system with
3 // productions X -> X+YF+, Y -> -FX-Y, initial word X
4 // and rotation angle 90 degrees
5 #include <iostream>
6 #include <IFM/turtle>
7 8 // necessary: x and y call each other
9 void y (const unsigned int i);
10 11 // POST: w_i^X is drawn
void x (const unsigned int i) {
  if (i > 0) {
    x(i-1); // \textit{w}_{i-1}^X
    ifm::left(90); // +
    y(i-1); // \textit{w}_{i-1}^Y
    ifm::forward(); // \textit{F}
    ifm::left(90); // +
  }
}

// POST: w_i^Y is drawn

void y (const unsigned int i) {
  if (i > 0) {
    ifm::right(90); // -
    ifm::forward(); // \textit{F}
    x(i-1); // \textit{w}_{i-1}^X
    ifm::right(90); // -
    y(i-1); // \textit{w}_{i-1}^Y
  }
}

int main () {
  std::cout << "Number of iterations =? ";
  unsigned int n;
  std::cin >> n;
  // draw w_n = w_n^X
  x(n);
  return 0;
}

Program 32: 

Finally, one can add symbols with graphical interpretation. Commonly used symbols are \texttt{f} (jump one step forward, this doesn't leave a trace), \texttt{[} (remember current position) and \texttt{]} (jump back to last remembered position). Our turtle library understands these three symbols as well. It is also typical to add new symbols with the same interpretation as \texttt{F}, say.

3.2.8 Details

Lindenmayer systems. Lindenmayer systems are named after the Danish biologist Aristide Lindenmayer (1925–1985) who proposed them in 1968 to model the growth of plants. Lindenmayer systems (with generalizations to 3-dimensional space) have found many
applications in computer graphics.

### 3.2.9 Goals

**Dispositional.** At this point, you should . . .

1) understand the concept of recursion, and why it makes sense to define a function through itself;
2) understand the semantics of recursive function calls and be aware that they do not always terminate;
3) appreciate the power of recursion in sorting and drawing Lindenmayer systems.

**Operational.** In particular, you should be able to . . .

(G1) find pre- and postconditions for given recursive functions;
(G2) prove or disprove termination and correctness of recursive function calls;
(G3) translate recursive mathematical function definitions into C++ function definitions;
(G4) rewrite a given recursive function in iterative form;
(G5) recognize inefficient recursive functions and improve their performance;
(G6) count the number of operations of a given type in a recursive function call, using induction as the main tool;
(G7) write recursive functions for given tasks.

### 3.2.10 Exercises

**Exercise 112** *Find pre- and postconditions for the following recursive functions.* (G1)

a) bool f (const int n)
   {
     if (n == 0) return false;
     return !f(n-1);
   }

b) void g (const unsigned int n)
   {
     if (n == 0) {
       std::cout << "*";
       return;
     }
     g(n-1);
     g(n-1);
   }
c) unsigned int h (const unsigned int n, const unsigned int b) {
    if (n == 1) return 0;
    return 1 + h(n / b, b);
}

Exercise 113 Prove or disprove for any of the following recursive functions that it terminates for all possible arguments. In this theory exercise, overflow should not be taken into account, i.e. you should pretend that the value range of unsigned int is equal to N.

a) unsigned int f (const unsigned int n)
{
    if (n == 0) return 1;
    return f(f(n-1));
}

b) // POST: return value is the Ackermann function value A(m,n)
unsigned int A (const unsigned int m, const unsigned int n) {
    if (m == 0) return n + 1;
    if (n == 0) return A(m - 1, 1);
    return A(m - 1, A(m, n - 1));
}

c) unsigned int f (const unsigned int n, const unsigned int m)
{
    if (n == 0) return 0;
    return 1 + f((n + m) / 2, 2 * m);
}

Exercise 114 Consider the following recursive function defined on all nonnegative integers, also known as McCarthy's 91 Function.

\[
M(n) := \begin{cases} 
  n - 10, & \text{if } n > 100 \\
  M(M(n + 11)), & \text{if } n \leq 100.
\end{cases}
\]

a) Provide a C++ function mccarthy that implements McCarthy's 91 Function.

b) What are the values of the following four function calls?

(i) mccarthy(101)
(ii) mccarthy(100)
(iii) mccarthy(99)
(iv) mccarthy(91)
3.2. RECURSION

(c) Explain why the function is called McCarthy’s 91 Function! More precisely, what is the value of M(n) for any given number n?

Exercise 115

(a) Write and test a C++ function that computes binomial coefficients \( \binom{n}{k} \), \( n, k \in \mathbb{N} \). These may be defined in various equivalent ways. For example,

\[
\binom{n}{k} := \frac{n!}{k!(n-k)!},
\]

or

\[
\binom{n}{k} := \begin{cases} 
0, & \text{if } n < k \\
1, & \text{if } n = k \text{ or } k = 0 \\
\binom{n-1}{k} + \binom{n-1}{k-1}, & \text{if } n > k, k > 0
\end{cases}
\]

or

\[
\binom{n}{k} := \begin{cases} 
0, & \text{if } n < k \\
1, & \text{if } n \geq k, k = 0 \\
\frac{n}{k} \binom{n-1}{k-1}, & \text{if } n \geq k, k > 0
\end{cases}
\]

(b) Which of the three variants is best suited for the implementation, and why? Argue theoretically, but underpin your arguments by comparing at least two different implementations of the function.

Exercise 116

In how many ways can you own CHF 1? Despite its somewhat philosophical appearance, the question is a mathematical one. Given some amount of money, in how many ways can you partition it using the available denominations (bank notes and coins)? Today’s denominations in CHF are 1000, 200, 100, 50, 20, 10 (banknotes), 5, 2, 1, 0.50, 0.20, 0.10, 0.05 (coins). The amount of CHF 0.20, for example, can be owned in four ways (to get integers, let’s switch to centimes): (20), (10, 10), (10, 5, 5), (5, 5, 5, 5). The amount of CHF 0.04 can be owned in no way, while there is exactly one way to own CHF 0.00 (you cannot have 4 centimes in your wallet, but you can have no money at all in your wallet).

Solve the problem for any given input amount, by writing a program partition.cpp that defines the following function (all values to be understood as centimes).
CHAPTER 3. FUNCTIONS

// PRE: [first, last) is a valid nonempty range that describes
// a sequence of denominations d_1 > d_2 > ... > d_n > 0
// POST: return value is the number of ways to partition amount
// using denominations from d_1, ..., d_n
unsigned int partitions (unsigned int amount,
    const unsigned int* first,
    const unsigned int* last);

Use your program to determine in how many ways you can own CHF 1, and CHF 10. Can your program compute the number of ways for CHF 50? For CHF 100?

Exercise 117 Suppose you want to crack somebody’s secret code, consisting of d digits between 1 and 9. You have somehow found out that exactly k of these digits are 1’s.

a) Write a program that generates all possible codes. The program should contain
   a function that solves the problem for given arguments d and k.

b) Adapt the program so that it also outputs the number of possible codes.

For example, if d = 2 and k = 1, the output may look like this:

12 13 14 15 16 17 18 19 21 31 41 51 61 71 81 91
There were 16 possible codes.

Exercise 118 Rewrite the following recursive function in iterative form and test with
a program whether your iterative version is correct. What can you say about the
absolute runtimes of both variants for values of n up to 100, say?

unsigned int f (const unsigned int n)
{
    if (n <= 2) return 1;
    return f(n-1) + 2 * f(n-3);
}

Exercise 119 Rewrite the following recursive function in iterative form and test with
a program whether your iterative version is correct. What can you say about the
runtimes of both variants for values of n up to 100, say?

unsigned int f (const unsigned int n)
{
    if (n == 0) return 1;
    return f(n-1) + 2 * f(n/2);
}
Exercise 120 Modify Program [29] (merge_sort.cpp) such that it calls the sorting function std::sort of the standard library instead of the function merge_sort. For this, you need the header algorithm that is already being included. Store the resulting program as std_sort.cpp and compare the total runtimes of both programs on larger inputs on your platform (start with \( n = 1,600,000 \) numbers and then keep doubling the input size).

What do you observe? How many sec/Gcomp does merge_sort take? Can you reproduce the "25" of Table 8, or do you get a different constant (explained by your computer having a speed different from the one of the authors)? How much faster (in absolute runtime) is std::sort on your platform? And is the speedup independent of \( n \)?

Exercise 121 The following function finds an element with a given value \( x \) in a sorted sequence (if there is such an element).

```cpp
// PRE: [first, last) is a valid range, and the elements *p, // p in [first, last) are in ascending order
// POST: return value is a pointer p in [first, last) such // that *p = x, or the pointer last, if no such pointer // exists
const int* binary_search (const int* first, const int* last, const int x)
{
    const int n = last - first;
    if (n == 0) return last; // empty range
    if (n == 1) {
        if (*first == x)
            return first;
        else
            return last;
    }
    // n >= 2
    const int* middle = first + n/2;
    if (*middle > x) {
        // x can't be in [middle, last)
        const int* p = binary_search (first, middle, x);
        if (p == middle)
            return last; // x not found
        else
            return p;
    } else
    // *middle <= x; we may skip [first, middle)
    return binary_search (middle, last, x);
}
```

What is the maximum number \( T(n) \) of comparisons between sequence elements and \( x \) that this function performs if the number of sequence elements is \( n \)? Try
to find an upper bound on $T(n)$ that is as good as possible. (You may use the statement of Exercise 122.) (G6)

Exercise 122 For any natural number $n \geq 2$, prove the following two (in)equalities. (G6)

$$\lceil \log_2 \left( \frac{n}{2} \right) \rceil \leq \lceil \log_2 \left( \frac{n}{2} \right) \rceil = \lceil \log_2 n \rceil - 1.$$

Exercise 123 Write programs that produce turtle graphics drawings for the following Lindenmayer systems $(\Sigma, P, s)$. (G7)

a) $\Sigma = \{F, +, -\}$, $s = F + F + F + F$ and $P$ given by

$$F \mapsto FF + F + F + F + F - F.$$

b) $\Sigma = \{X, Y, +, -\}$, $s = Y$, and $P$ given by

$$X \mapsto Y + X + Y$$
$$Y \mapsto X - Y - X.$$

For the drawing, use rotation angle $\alpha = 60$ degrees and interpret both $X$ and $Y$ as “move one step forward”.

c) Like b), but with the productions

$$X \mapsto X + Y + +Y - X - -XX - Y +$$
$$Y \mapsto -X + YY + +Y + X - -X - Y.$$

Exercise 124 The Towers of Hanoi puzzle (that can actually be bought from shops) is the following. There are three wooden pegs labeled 1, 2, 3, where the first peg holds a stack of $n$ disks, stacked in decreasing order of size, see Figure Figure 22.

The goal is to transfer the stack of disks to peg 3, by moving one disk at a time from one peg to another. The rule is that at no time, a larger disk may be on top of a smaller one. For example, we could start by moving the topmost disk to peg 2 (move (1, 2)), then move the next disk from peg 1 to peg 2 (move (1, 3)), then move the smaller disk from peg 2 onto the larger disk on peg 3 (move (2, 3)), etc.

Write a program hanoi.cpp that outputs a sequence of moves that does the required transfer, for given input $n$. For example, if $n = 2$, the above initial sequence (1, 2)(1, 3)(2, 3) is already complete and solves the puzzle. Check the correctness of your program by hand for at least for $n = 3$, by manually reproducing the sequence of moves on a piece of paper (or an actual Tower of Hanoi, if you have one). (G7)
3.2. RECURSION

3.2.11 Challenges

Exercise 125 This challenge lets you take a look behind the scenes of GIMPS (the Great Internet Mersenne Prime Search, www.mersenne.org). This project is concerned with the problem of finding large Mersenne primes, see Section 1.1. The essential tool used by GIMPS is the Lucas-Lehmer test, a simple procedure to check whether a number of the form $2^n - 1$ is prime or not.

Find out what the Lucas-Lehmer test is and write a program that employs it to find all Mersenne primes of the form $2^n - 1$ with $n \leq 1,000$ (you may go higher if you can). The resulting list will allow you to rediscover the five errors that Mersenne made in his original conjecture (Section 1.1), and it will tell you what happens next after $n = 257$, the largest exponent covered by Mersenne’s conjecture.

Exercise 126 On the occasion of major sports events, the Italian company Panini sells stickers to be collected in an album. For the EURO 2008 soccer championship, the collection comprised of 555 different stickers, available in packages of five stickers each.

When buying a package, you cannot see which stickers it contains. The company only guarantees that each package contains five different stickers. Let us assume that each possible selection of five different stickers is equally likely to be contained in any given package. How many packages do you need to buy on average in order to have all the stickers?

For the case of EURO 2008 with 555 stickers, a newspaper claimed (based on consulting a math professor) that this number is 763. How did the professor arrive at that number, and is it correct?

Write a program that computes the average number of packages that you need to buy for a collection of size $n$. (As a simple check, you should get one package on average if $n = 5$). What do you get for $n = 555$?

Note: In order to solve this challenge in a mathematically sound way, you need some basic knowledge of probability theory. But for our purposes, it is also ok to just handwave why your program is correct.
Exercise 127 Lindenmayer systems can also be used to draw (quite realistic) plants, with the growth process simulated by the various iterations. For this, however, there must be a possibility of creating branches. Let us therefore enhance our default set \{F, +, −\} of symbols with fixed meaning and now use  \(\Sigma = \{F, +, −, [,], f\}\). The symbol [ is defined to have the following effect: the current state of the turtle (position and direction) is put on top of the state stack which is initially empty. The symbol ] sets the state of the turtle back to the one found on top of the state stack, and removes the top state from the stack. This mechanism can be used to remember a certain state and return to it later.

For example, if the rotation angle is 45 degrees, the word FF+[F][−F] produces the drawing of Figure 23.

![Figure 23: The turtle after processing the command sequence FF+[F][−F]](image)

This does not look like a very sophisticated plant yet, but if you for example try the production

\[ F \rightarrow FF + [+F − F − F] − [−F + F + F] \]

with initial word F, rotation angle 22 degrees, and four iterations, you will see what we mean.

It remains to explain what the symbol f means. It has the same effect on the state of the turtle as F, except that it does not draw a line. You can imagine that f makes the turtle “jump”.

Here are the functions of the library turtle that correspond to this additional functionality. jump realizes f, while save and restore are for [ and ]. In order to draw Figure 23, we can therefore use the following statements.

```c
ifm::forward(2);
ifm::save();
ifm::left(45);
ifm::forward();
ifm::restore();
ifm::save();
ifm::right(45);
```
3.2. RECURSION

ifm::forward();
ifm::restore();

Here you see that we can provide an integer to forward telling it how many steps we want to move forward (the default that we always used before is 1).

Now here comes the challenge: write a turtle graphics program amazing.cpp that will knock our socks off! In other words, we are asking for the most beautiful picture that you can produce using the recursive drawing scheme on top of the turtle graphics commands introduced so far (there are still more commands that are more or less common, but our turtle library stops at $\Sigma = \{F, +, -, [ ], f\}$).

If you think that you can submit a crappy program and still earn full points, you're right. But we count on your sportsmanship to give your best!
Chapter 4

Compound Types
4.1 Structs

A POD-struct is an aggregate class that has no nonstatic data members of type pointer to member, non-POD struct, non-POD union (or array of such types) or reference, and has no user-defined copy-assignment operator and no user-defined destructor.

Section 9, paragraph 4, of the ISO/IEC Standard 14882 (C++ Standard)

In this section, we show how structs are used to group data and to obtain new types with application-specific functionality. You will also see how operator overloading can help in making new types easy and intuitive to use.

Suppose we want to use rational numbers in a program, i.e., numbers of the form \( \frac{n}{d} \), where both the numerator \( n \) and the denominator \( d \) are integers. C++ does not have a fundamental type for rational numbers, so we have to implement it ourselves.

We could of course represent a rational number simply by two values of type \( \text{int} \), but this would not be in line with our perception of the rational numbers as a distinct mathematical concept. The two numbers \( n \) and \( d \) "belong together", and this is also reflected in mathematical notation: the symbol \( \mathbb{Q} \) for the set of rational numbers indicates that we are dealing with a mathematical type, defined by its value range and its functionality (see Section 2.1.6). Ideally, we would like to get a C++ type that can be used like existing arithmetic types; the following piece of code (for adding two rational numbers) shows how this could look like.

```cpp
// input
std::cout << "Rational number r:\n";
raional r;
std::cin >> r;

std::cout << "Rational number s:\n";
raional s;
std::cin >> s;

// computation and output
std::cout << "Sum is " << r + s << ".\n";
```

And we would like a sample run of this to look as follows.

Rational number r:
1/2
Rational number s:
In C++ this can be done. C++ offers the possibility of defining new types based on existing types. In this section, we introduce as a first step the concept of structs. A struct is used to aggregate several values of different types into one value of a new type. With this, we can easily model the mathematical type \( \mathbb{Q} \) as a new type in C++. Here is a working program that makes a first step toward the desired piece of code above.

```cpp
// Program: userational.cpp
// Add two rational numbers.
#include <iostream>

// the new type rational
struct rational {
    int n;
    int d; // INV: d != 0
};

// POST: return value is the sum of a and b
rational add (const rational a, const rational b) {
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}

int main () {
    // input
    std::cout << "Rational number r:\n";
    rational r;
    std::cout << " numerator =? " ; std::cin >> r.n;
    std::cout << " denominator =? " ; std::cin >> r.d;

    std::cout << "Rational number s:\n";
    rational s;
    std::cout << " numerator =? " ; std::cin >> s.n;
    std::cout << " denominator =? " ; std::cin >> s.d;

    // computation
    const rational t = add (r, s);

    // output
    std::cout << "Sum is " << t.n << "/" << t.d << ".\n";
```
In C++, a struct defines a new type whose value range is the Cartesian product of a fixed number of types. (Here and in the following, we identify a type with its value range to avoid clumsy formulations.) In our case, we define a new type named rational whose value range is the Cartesian product \( \text{int} \times \text{int} \), where we interpret a value \((n, d)\) as the quotient \(n/d\).

Since there is no type for the denominator with the appropriate value range \(\text{int}\setminus\{0\}\), we specify the requirement \(d \neq 0\) by an informal invariant, a condition that has to hold for all legal combinations of values. Such an invariant is indicated by a comment starting with

```
// INV:
```

Like pre- and postconditions of functions (see Section 3.1.1), invariants are an informal way of documenting the program; they are not standardized, and our way of writing them is one possible convention.

The type rational is referred to as a struct, and it can be used like any other type; for example, it may appear as argument type and return type in functions like add.

A struct defines a type, not variables. Let's get rid of one possible confusion right from the beginning. The definition

```
struct rational {
    int n;
    int d; // INV: d \neq 0
};
```

does not define variables \(n\) and \(d\) of type int, although the two middle lines look like variable declarations as we know them. Rather, all four lines together define a type of the name rational, but at that point, neither a variable of that new type, nor variables of type int have been defined. The two middle lines

```
int n;
int d; // INV: d \neq 0
```

specify that every actual object of the new type (i.e. a concrete rational number) “has” (is represented by) two objects of type int that can be accessed through the names \(n\) and \(d\); see the member access below. This specification is important if we want to implement operations on our new type like in the function add.

Here is an analogy for the situation. If the university administration wants to specify how a student is represented in their files, they might come up with three pieces of data that are necessary: a name, an identification number, and a program of study. This
4.1. STRUCTS

defines the “type” of a student and allows functionality (registration, change of program of study, etc.) to be realized, long before any students actually show up.

4.1.1 Struct definitions.

In general, a struct definition looks as follows.

```c
struct T {
    T1 name1;
    T2 name2;
    ...
    TN nameN;
};
```

Here, $T$ is the name of the newly introduced struct (this name must be an identifier, Section 2.1.10), and $T_1, \ldots, T_N$ are names of existing types. These are called the underlying types of $T$. The identifiers $name_1, name_2, \ldots, name_N$ are the data members of the new type $T$.

The value range of $T$ is $T_1 \times T_2 \times \ldots \times T_N$. This means, a value of type $T$ is an $N$-tuple $(t_1, t_2, \ldots, t_N)$ where $t_i \in T_i$.

“Existing types” might be fundamental types, but also user-defined types. For example, consider the vector space $\mathbb{Q}^3$ over the field $\mathbb{Q}$. Given the type rational as above, we could model $\mathbb{Q}^3$ as follows.

```c
struct rational_vector_3 {
    rational x;
    rational y;
    rational z;
};
```

Although it follows from the definition, let us make it explicit: the types $T_1, \ldots, T_N$ need not be the same. Here is an example: If $0, 1, \ldots, U$ is the value range of the type unsigned int, we can get a variant of the type int with value range

$\{-U, -U + 1, \ldots, -1, 0, 1, \ldots, U - 1, U\}$

as follows.

```c
struct extended_int {
    // represents u if n==false and -u otherwise
    unsigned int u; // absolute value
    bool n; // sign bit
};
```

The value range of this type is $\{0, 1, \ldots, U\} \times \{\text{true, false}\}$, but like in the rational case, we interpret values differently: a value $(u, n)$ “means” $u$ if $n = \text{false}$ and $-u$ if $n = \text{true}$. 
Even if two struct definitions have the same member specification (the part of the definition enclosed in curly braces), they define different types, and it is not possible to replace one for the other. Consider this trivial but instructive example with two apparently equal structs defined over an empty set of existing types.

```c
struct S {
};

struct T {
};

void foo (const S s) {};

int main () {
    S s;
    T t;
    foo (s); // ok
    foo (t); // error: type mismatch
    return 0;
}
```

It is also possible to use array members in structs. For example, the field $Q^3$ that we have discussed above could alternatively be modeled like this.

```c
struct rational_vector_3 {
    rational v[3];
};
```

### 4.1.2 Structs and scope

The scope of a struct is the part of the program in which it can be used (in a variable declaration, or as a formal function argument type, for example). Structs behave similar to functions here: the scope of a struct is the union of the scopes of all its declarations, where a struct declaration has the form

```c
struct T
```

The struct definition is a declaration as well, and usually one actually needs the definition in order to use a struct. This is easy to explain: in order to translate a variable declaration of struct type, or a function with formal arguments of struct type into machine language, the compiler needs to know the amount of memory required by an object of the struct. But this information is only obtainable from the definition of the struct; as long as the compiler has only seen a declaration of $T$, the struct $T$ is said to have incomplete type.
4.1.3 Member access

A struct is more than the Cartesian product of its underlying types—it offers some basic functionality on its own that we explain next. The most important (and also most visible) functionality of a struct is the access to the data members, and here is where the identifiers name1,..., nameN come in. If expr is an expression of type T with value (t1,...,tN), then tK—the K-th component of its value—can be accessed as

\[ expr.nameK \]

Here, ‘.’ is the member access operator (see Table 9 in the Appendix for its specifics). The composite expression expr.nameK is an lvalue if expr itself is an lvalue, and we say that the data member nameK is accessed for expr.

Lines 25 and 26 of Program 33 assign values to the rational numbers r through the member access operator, while line 37 employs the member access operator to output the value of the rational number t. The additional output of ‘/’ indicates that we interpret the 2-tuple (n, d) as the quotient n/d.

4.1.4 Initialization and assignment

We can initialize objects of struct type and assign values to them, just like we do it for fundamental types.

In line 34 of Program 33 for example, the variable t of type rational is initialized with the value of the expression add (r, s). In a struct, initialization is quite naturally done member-wise, i.e. for each data member separately. Under the hood, the declaration statement

\[ \text{const rational } t = \text{add } (r, s); \]

therefore has the effect of initializing t.n (with the first component of the value of add (r, s)) and t.d (with the second component). Interestingly, this also works with array members. Structs therefore provide a way of forging array initialization and assignment by wrapping the array into a struct. Here is an example to show what we mean.

```c
#include <iostream>

struct point {
    double coord[2];
};

int main()
{
    point p;
    p.coord[0] = 1;
    p.coord[1] = 2;
```
point q = p;
std::cout << q.coord[0] << " "  // 1
  << q.coord[1] << "\n";  // 2
return 0;
}

In the same way (memberwise initialization), the formal arguments \( a \) and \( b \) of the function \( \text{add} \) are initialized from the values of \( r \) and \( s \); the value of \( \text{add} (r, s) \) itself also results from an initialization of a (temporary) object when the return statement of the function \( \text{add} \) is executed.

Instead of the above declaration statement that initializes \( t \) we could also have written

```c++
    rational t;
    t = add (r, s);
```

Here, \( t \) is default-initialized first, and this default-initializes the data members. In our case, they are of type \( \text{int} \); for fundamental types, default-initialization does nothing, so the values of the data members are undefined after default-initialization (see also Section 2.1.8). In the next line, the value of \( \text{add} (r, s) \) is assigned to \( t \), and this assignment again happens member-wise.

What about other operations? For every fundamental type \( T \), two expressions of type \( T \) can be tested for equality, using the operators \( == \) and \( != \). It would therefore seem natural to have these two operators also for structs, implemented in such a way that they compare member-wise.

Formally, this would be correct: if \( t = (t_1, \ldots, t_N) \) and \( t' = (t'_1, \ldots, t'_N) \), then we have \( t = t' \) if and only if \( t_K = t'_K \) for \( K = 1, \ldots, N \).

But our type \( \text{rational} \) already shows that this won't work: under member-wise equality, we would erroneously conclude that \( 2/3 \neq 4/6 \). The problem is that the syntactical value range \( \text{int} \times \text{int} \) of the type \( \text{rational} \) does not coincide with the semantical value range in which we identify pairs \( (n, d) \) that define the same rational number \( n/d \).

The same happens with our type \( \text{extended} \_\text{int} \) from above: since both pairs \((0, \text{false})\) and \((0, \text{true})\) are interpreted as \( 0 \), member-wise equality would give us "\( 0 \neq 0 \)" in this case.

Only the implementor of a struct knows the semantical value range, and for this reason, C++ neither provides equality operators for structs, nor any other operations beyond the member access, initialization, and assignment discussed above. Operations that respect the semantical value range can be provided by the implementor, though, see next section.

You might argue that even member-wise initialization and assignment could be inconsistent with the semantics of the type. Later, we will indeed encounter such a situation,
and we will show how it can be dealt with elegantly.

4.1.5 User-defined operators

New types require new operations, but when it comes to the naming of such operations, one less nice aspect of Program 33 shows in line 34. By defining the function add, we were able to perform the operation \( t := r + s \) through the statement

\[
\text{const rational } t = \text{add} (r, s);
\]

Ideally, however, we would like to add rational numbers like we add integers or floating-point numbers, by simply writing (in our case)

\[
\text{const rational } t = r + s;
\]

The benefit of this might not be immediately obvious, in particular since the naming of the function add seems to be quite reasonable; but consider the expression

\[
\text{const rational } t = \text{subtract} (\text{multiply} (p, q), \text{multiply} (r, s));
\]

and its "natural" counterpart

\[
\text{const rational } t = p * q - r * s;
\]

to get an idea what we mean.

The natural notation can indeed be achieved: a key feature of the C++ language is that most of its operators (see Table 9 in the Appendix for the full list) can be overloaded to work for other types as well. This means that we can use the same operator token to implement various operators: we "overload" the token.

In principle, this is nothing new: we already know that the binary operator + is available for several types, for example int and double. What is new is that we can add even more overloads on our own, and simply let the compiler figure out from the call argument types which one is needed in a certain context.

In overloading an operator, we cannot change the operator's arity, precedence or associativity, but we can create versions of it with arbitrary formal argument and return types.

Operator overloading is simply a special case of function overloading. For example, having the structs rational and extended_int available, we could declare the following two functions in the same program, without creating a name clash: for any call to the function square in the program, the compiler can find out from the call argument type which of the two functions we mean.

// POST: returns a * a
rational square (rational a);

// POST: returns a * a
extended_int square (extended_int a);
Function overloading in general is useful, but not nearly as useful as operator overloading. To define an overloaded operator, we have to use the functional operator notation. In this notation, the name of the operator is obtained by appending its token to the prefix operator. In case of the binary addition operator for the type rational, this looks as follows and replaces the function add.

```cpp
// POST: return value is the sum of a and b
rational operator+ (const rational a, const rational b) {
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
```

In Program [33] we can now replace line 34 by

```cpp
const rational t = r + s; // equivalent to operator+ (r, s);
```

Here, the comment refers to the fact that an operator can also be called in functional notation; in contrast, it appears in infix notation in \( r + s \). The call in functional notation can be useful for didactic purposes, since it emphasizes the fact that an operator is simply a special function; in an application, however, the point is to avoid functional notation and use the infix notation.

The other three basic arithmetic operations are similar, and here we only give their declarations.

```cpp
// POST: return value is the difference of a and b
rational operator- (rational a, rational b);

// POST: return value is the product of a and b
rational operator* (rational a, rational b);

// POST: return value is the quotient of a and b
// PRE: b != 0
rational operator/ (rational a, rational b);
```

We can also overload the unary - operator; in functional operator notation, it has the same name as the binary version, but it has only one instead of two arguments. In the following implementation, we use the (modified) "local copy" of the call argument \( a \) as the return value.

```cpp
// POST: return value is -a
rational operator- (rational a) {
    a.n = -a.n;
    return a;
}
```
In order to compare rational numbers, we need the relational operators as well. Here is the equality operator as an example.

```cpp
// POST: return value is true if and only if a == b
bool operator==(const rational a, const rational b)
{
    return a.n * b.d == a.d * b.n;
}
```

### 4.1.6 Details

**Overloading resolution.** If there are several functions or operators of the same name in a program, the compiler has to figure out which one is meant in a certain function call. This process is called *overloading resolution* and only depends on the types of the call arguments. Overloading resolution is therefore done at compile time. There are two cases that we need to consider: we can either have an *unqualified* function call (like `add(r, s)` in Program 33), or a *qualified* function call (like `std::sqrt(2.0)`). To process an unqualified function call of the form `fname(expression1, ..., expressionN)` the compiler has to find a matching function declaration. Candidates are all functions `f` of name `fname` such that the function call is in the scope of some declaration of `f`. In addition, the number of formal arguments must match the number of call arguments, and each call argument must be of a type whose values can be converted to the corresponding formal argument types.

In a qualified function call of the form `X::fname(expression1, ..., expressionN)` where `X` is a namespace, only this namespace is searched for candidates.

**Argument-dependent name lookup (Koenig lookup).** There is one special rule that sometimes makes the list of candidates larger. If some call argument type of an unqualified function call is defined in a namespace `X` (for example the namespace `std`), then the compiler also searches for candidates in `X`. This is useful mainly for operators and allows them to be called unqualified in infix notation. The point of using operators in infix notation would be spoiled if we had to mention a namespace somewhere in the operator call.

**Resolution: Finding the best match.** For each candidate function and each call argument, it is checked how well the call argument type matches the corresponding formal argument type. There are four quality levels, going from better to worse, given in the following list.
(1) **Exact match.** The types of the call argument and the formal argument are the same.

(2) **Promotion match.** There is a promotion from the call argument type to the formal argument type. We have seen some examples for promotions, like from `bool` to `int` and from `float` to `double`.

(3) **Standard conversion match.** There is a standard conversion from the call argument type to the formal argument type. We have seen that all fundamental arithmetic types can be converted into each other by standard conversions.

(4) **User-defined conversion match.** There is a user-defined conversion from the call argument type to the formal argument type. We will get to user-defined conversions only later in this book.

A function \( f \) is called **better** than \( g \) with respect to an argument, if the match that \( f \) induces on that argument is at least as good as the match induced by \( g \). If the match is really better, \( f \) is called **strictly better** for the argument.

A function \( f \) is called a **best match** if it is better than any other candidate \( g \) in all arguments, and strictly better than \( g \) in at least one argument.

Under this definition, there is at most one best match, but it may happen that there is no best match, in which case the function call is **ambiguous**, and the compiler issues an error message.

Here is an example. Consider the two overloaded function declarations

```c++
void foo(double d);
void foo(unsigned int u);
```

In the code fragment

```c++
float f = 1.0f;
foo(f);
```

the first overload is chosen, since `float` can be promoted to `double`, but only standard-converted to `unsigned int`. In

```c++
int i = 1;
foo(i);
```

the call is ambiguous, since `int` can be standard-converted to both `double` and `unsigned int`.

### 4.1.7 Goals

**Dispositional.** At this point, you should...

1) know how structs can be used to aggregate several different types into one new type;

2) understand the difference between the syntactical and semantical value range of a struct;

3) know that C++ functions and operators can be overloaded.
4.1. STRUCTS

Operational. In particular, you should be able to ...

(G1) define structs whose semantical value ranges correspond to that of given mathematical sets;

(G2) provide definitions of functions and overloaded operators on structs, according to given functionality;

(G3) write programs that define and use structs according to given functionality.

4.1.8 Exercises

Exercise 128 A struct may have data members of struct type that may have data members of struct type, and so on. It can be quite cumbersome to set all the data members of the data members of the data members manually whenever an object of the struct type needs to be initialized. A preferable solution is to write a function that does this. As an example, consider the type

```c
struct rational_vector_3 {
    rational x;
    rational y;
    rational z;
};
```

where `rational` is as before defined as

```c
struct rational {
    int n;
    int d;  // INV: d != 0
};
```

Write a function

```c
rational_vector_3 create_rational_vector_3
    (int n1, int d1, int n2, int d2, int n3, int d3)
```

that returns the rational vector \((n_1/d_1, n_2/d_2, n_3/d_3)\). The function should also make sure that \(d_1, d_2, d_3\) are nonzero. (G3)

Exercise 129 Define a type `Tribool` for three-valued logic; in three-valued logic, we have the truth values true, false, and unknown.

For the type `Tribool`, implement the logical operators

```c
// POST: returns x AND y
Tribool operator && (Tribool x, Tribool y);

// POST: returns x OR y
Tribool operator || (Tribool x, Tribool y);
```
where AND (\( \land \)) and OR (\( \lor \)) are defined according to the following two tables.

\[
\begin{array}{ccc}
\land & \text{false} & \text{unknown} & \text{true} \\
\text{false} & \text{false} & \text{false} & \text{false} \\
\text{unknown} & \text{false} & \text{unknown} & \text{true} \\
\text{true} & \text{false} & \text{unknown} & \text{true} \\
\end{array}
\quad \begin{array}{ccc}
\lor & \text{false} & \text{unknown} & \text{true} \\
\text{false} & \text{false} & \text{unknown} & \text{true} \\
\text{unknown} & \text{unknown} & \text{unknown} & \text{true} \\
\text{true} & \text{true} & \text{true} & \text{true} \\
\end{array}
\]

Test your type by writing a program that outputs these truth tables in some format of your choice.

**Exercise 130** Define a type \( \mathbb{Z}_7 \) for computing with integers modulo 7. Mathematically, this corresponds to the finite ring \( \mathbb{Z}_7 = \mathbb{Z}/7\mathbb{Z} \) of residue classes modulo 7.

For the type \( \mathbb{Z}_7 \), implement addition and subtraction operators

```cpp
// POST: return value is the sum of a and b
std::string Z_7 operator + ( Z_7 a, Z_7 b);

// POST: return value is the difference of a and b
std::string Z_7 operator - ( Z_7 a, Z_7 b);
```

according to the following table (this table also defines subtraction: \( x - y \) is the unique number \( z \in \{0, \ldots, 6\} \) such that \( x = y + z \)).

\[
\begin{array}{cccccccc}
+ & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
0 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
1 & 1 & 2 & 3 & 4 & 5 & 6 & 0 \\
2 & 2 & 3 & 4 & 5 & 6 & 0 & 1 \\
3 & 3 & 4 & 5 & 6 & 0 & 1 & 2 \\
4 & 4 & 5 & 6 & 0 & 1 & 2 & 3 \\
5 & 5 & 6 & 0 & 1 & 2 & 3 & 4 \\
6 & 6 & 0 & 1 & 2 & 3 & 4 & 5 \\
\end{array}
\]

**Exercise 131** Provide definitions for the following binary arithmetic operators on the type rational.

```cpp
// POST: return value is the difference of a and b
std::string rational operator - ( rational a, rational b);

// POST: return value is the product of a and b
std::string rational operator * ( rational a, rational b);

// POST: return value is the quotient of a and b
// PRE: b != 0
std::string rational operator / ( rational a, rational b);
```
Exercise 132 Provide definitions for the following binary relational operators on the type rational. In doing this, try to reuse operators that are already defined.

```cpp
bool operator!= (rational a, rational b);
// POST: return value is true if and only if a != b

bool operator< (rational a, rational b);
// POST: return value is true if and only if a < b

bool operator<= (rational a, rational b);
// POST: return value is true if and only if a <= b

bool operator> (rational a, rational b);
// POST: return value is true if and only if a > b

bool operator>= (rational a, rational b);
// POST: return value is true if and only if a >= b
```

Exercise 133 Provide definitions for the following binary arithmetic operators on the type extended_int (Page 249), and test them in a program (for that it could be helpful to provide an output facility for the type extended_int, and a function that assigns to an extended_int value a value of type int). As in the previous exercise, try to reuse code.

```cpp
extended_int operator+ (extended_int a, extended_int b);
// POST: return value is the sum of a and b

extended_int operator- (extended_int a, extended_int b);
// POST: return value is the difference of a and b

extended_int operator* (extended_int a, extended_int b);
// POST: return value is the product of a and b

extended_int operator- (extended_int a);
// POST: return value is -a
```

Exercise 134 Consider the following set of three functions.

```cpp
void foo(double, double) { ... } // function A
void foo(unsigned int, int) { ... } // function B
void foo(float, unsigned int) { ... } // function C
```

For each of the following function calls, decide to which of the functions (A,B,C) it resolves to, or decide that the call is ambiguous. Explain your decisions! This exercise requires you to read the paragraph on overloading resolution in the Details section.
4.1.9 Challenges

Exercise 135 This challenge has a computer graphics flavor. Write a program that allows you to visualize and manipulate a 3-dimensional object. For the sake of concreteness, think of a wireframe model of a cube given by 12 edges in threedimensional space.

The program should be able to draw the object in perspective view and at least provide the user with a possibility of rotating the object around the three axes. The drawing window might for example look like in Figure 24.

Instead of a cube, you may want to take another platonic solid, you may read the wireframe model from a file, you may add the possibility of scaling the object, translating it, etc. Use the library libwindow that is available at the course homepage to create the graphical output.

If you don’t know (or have forgotten) how to rotate and project threedimensional points, here is a crash course.

Rotating a point \((x, y) \in \mathbb{R}^2\) around the origin by an angle of \(\alpha\) (radians) results in the point \((x', y')\) with coordinates

\[
\begin{pmatrix}
    x' \\
    y'
\end{pmatrix} = \begin{pmatrix}
    \cos \alpha & -\sin \alpha \\
    \sin \alpha & \cos \alpha
\end{pmatrix} \begin{pmatrix}
    x \\
    y
\end{pmatrix}.
\]

In order to rotate a point \((x, y, z) \in \mathbb{R}^3\) around the z-axis, you simply keep z unchanged and rotate \((x, y)\) as above. By symmetry, you can figure out how this works for the other axes.

General perspective projection is not so easy, but if you want to project a point onto the \(z = 0\)-plane (imagine that this plane is the computer screen that you want to draw on), this is not hard. Imagine that \(v = (v_x, v_y, v_z)\) is the viewpoint (position of your eye). \(v_z > 0\) for example means that you are sitting in front of the screen. When you project the point \(p = (x, y, z)\) onto the screen, the image point has coordinates

\[
(x - t(v_x - x), y - t(v_y - y)),
\]
where
\[ t = \frac{z}{v_z - z}. \]

The projection thus only works if \( v_z \neq z \).
4.2 Reference Types

Whereas Europeans generally pronounce my name the right way ('Ni-klows Wirt'), Americans invariably mangle it into 'Nick-les Worth'. This is to say that Europeans call me by name, but Americans call me by value.

Attributed to Niklaus Wirth

This section explains reference types that enable functions to accept and return lvalues and in particular change the values of their call arguments. Reference types naturally come up in implementing functionality for structs.

4.2.1 The trouble with operator+=

Let us try to implement the addition assignment operator += for the struct rational from Program 33. Here is an attempt:

```cpp
rational operator+= (rational a, const rational b) {
    a.n = a.n * b.d + a.d * b.n;
    a.d *= b.d;
    return a;
}
```

With this, we can write

```cpp
rational r;
r.n = 1; r.d = 2; // 1/2

rational s;
s.n = 1; s.d = 3; // 1/3

r += s;
std::cout << r.n << "/" << r.d << "\n";
```

You may already see that the output of this will not be the desired 5/6. Recall from Section 3.1.3 what happens when r += s (equivalently, operator+= (r, s)) is evaluated: r and s are evaluated, and the resulting values are used to initialize the formal arguments a and b of the function operator+=. The values of r and s are not changed by the function call.

Hence, with the above implementation of operator+=, the value of the expression r += s is indeed 5/6, but the desired effect, the change of r's value, does not happen. That's why we get 1/2 as output in the above piece of code.
4.2. REFERENCE TYPES

In order to implement \texttt{operator+=} properly, we must enable functions to change the values of their call arguments. The careful reader may remember that pointer types (Section 2.6.6) can indirectly be used to achieve this, but with a somewhat clumsy syntax and potentially unsafe behavior due to pointers “going nowhere”, see Exercise 100.

Reference types are very similar to pointer types in their philosophy, but in most contexts, they are more natural to use and at the same time more safe, since they render the use of explicit (and possibly invalid) pointers unnecessary.

4.2.2 Reference Types: Definition

If \( T \) is any type, then

\[ T\& \]

is the corresponding reference type (read \( T\& \) as “\( T \) reference” or “reference to \( T \)”). In value range and functionality, \( T\& \) is identical to \( T \). The difference is only in the initialization and assignment semantics. An expression of reference type is called a reference.

A variable of reference type \( T\& \) can be initialized only from an \textit{lvalue} of type \( T \). The initialization makes it an alias of the lvalue: another name for the object behind the lvalue. We also say that the reference refers to that object. The following example shows this.

\begin{verbatim}
int i = 5;
int& j = i;       // j becomes an alias of i
j = 6;            // changes the values of i
std::cout << i << "\n"; // outputs 6
\end{verbatim}

Declarations such as

\begin{verbatim}
int& j;       // error: no object to refer to
int& k = 5;   // error: 5 is not an lvalue
\end{verbatim}

are invalid.

A reference behaves like (and is implemented) by a constant pointer to the object that it refers to, a pointer that is dereferenced whenever the reference is evaluated. But the pointer and its value—the address of the object—are hidden. Here is the explicit-pointer variant of the above code fragment that changes the value of \( i \) to 6:

\begin{verbatim}
int i = 5;
int* const j = &i;   // j points to i
*j = 6;              // changes the values of i
std::cout << i << "\n"; // outputs 6
\end{verbatim}

Apart from the syntax, the main difference between the two declarations
int & j = lvalue
and
int * const j = ptr
is that in the former case, the constant pointer hidden behind the reference j is guaranteed to represent a valid address, namely the one of lvalue. In the latter case, the explicit constant pointer j may or may not be valid, depending on whether the pointer ptr is valid. From a safety point of view, working with references is therefore preferable to working with pointers.

The fact that a reference is implemented by a constant pointer implies that a reference cannot be changed to refer to another object after initialization. If we later assign something to the reference, we in fact assign to the object referred to by it. In writing j = 6 in the above piece of code, we therefore change the value of i to 6, since j is an alias of i.

Every reference is an lvalue itself. We can therefore use a reference r to initialize another reference r', but then we don't get a reference to r, but another reference to the object referred to by r:

```cpp
int i = 5;
int & j = i;  // j becomes an alias of i
int & k = j;  // k becomes another alias of i
```

4.2.3 Call by value and call by reference

When a function has a formal argument of reference type, the corresponding call argument must be an lvalue; when the function call is evaluated, the initialization of the formal argument makes it an alias of the call argument. In this way, we can implement functions that change the values of their call arguments. Here is an example.

```cpp
void increment (int & i)
{
    ++i;
}

int main ()
{
    int j = 5;
    increment (j);
    std::cout << j << \n;  // outputs 6

    return 0;
}
```

If a formal argument of a function has reference type, we have call-by-reference semantics with respect to that argument. Equivalently, we say that we pass the argument by reference. Another frequently used term for call by reference is call by name.
If the formal argument is not of reference type, we have *call-by-value* semantics: we pass the argument by value. Under call by reference, the address of (or a reference to) the call argument is used to initialize the formal argument; under call-by-value semantics, it is the value of the call argument that is used for initialization.

The basic rule is to pass an argument by reference only if the function in question actually needs to change the call argument value. If that is not the case, call by value is more flexible, since it allows a larger class of call arguments (*lvalues* and *rvalues* instead of *lvalues* only).

### 4.2.4 Return by value and return by reference

The return type of a function can be a reference type as well, in which case we have *return-by-reference* semantics (otherwise, we *return by value*). If the function returns a reference, the function call expression is an *lvalue* itself, and we can use it wherever *lvalues* are expected.

This means that the function itself chooses (by using reference types or not) whether its call arguments and return value are *lvalues* or *rvalues*. Section 2.1.14 and Section 2.2.4 document these choices for some of the operators on fundamental types, but only now we understand the mechanism that makes such choices possible.

As a concrete example, let us consider the following version of the function `increment` that exactly models the behavior of the pre-increment operator `++`: it increments its *lvalue* argument and returns it as an *lvalue*.

```cpp
int & increment (int & i)
{
    return ++i;
}
```

In general, we must make sure that an expression of reference type that we return refers to a *non-temporary* object. To understand what a temporary object is, let us consider the following function.

```cpp
int & foo (const int i)
{
    return i;
}
```

This is asking for trouble, since the formal argument `i` runs out of scope when the function call terminates. This means that the associated memory is freed and the address expires (see Section 2.4.3). If we now write for example

```cpp
int i = 3;
int & j = foo(i);   // j refers to expired object
std::cout << j;    // undefined behavior
```

the reference `j` refers to an expired object, and the resulting behavior of the program is undefined. References are safer than pointers, but not absolutely safe. In working with references, we need to adhere to the following guideline.
Reference Guideline: Whenever you create an alias for an object, ensure that the object does not expire before the alias.

The compiler usually notices violations of the Reference Guideline and issues a warning.

4.2.5 More user-defined operators

Rational numbers: addition assignment. Let’s get back to the addition assignment operator for our new struct rational. In order to fix our failed attempt from the beginning of this section, we need to add two characters only.

As in the previous function increment, the formal argument a must be passed as a reference, and to be compliant with the usual semantics of +=, we also return the result as a reference:

```c++
// POST: b has been added to a; return value is the new value of a
rational& operator+=(rational& a, const rational b) {
    a.n = a.n * b.d + a.d * b.n;
    a.d *= b.d;
    return a;
}
```

The other arithmetic assignment operators are similar, and we don’t list them here explicitly. Together with the arithmetic and relational operators discussed in Section 4.1.5, we now have a useful set of operations on rational numbers.

Rational numbers: input and output. Let us look at Program 33 once more, with the function name add replaced by operator+ and the function call add (r, s) replaced by r + s. Still, we can spot potential improvements: instead of writing

```c++
std::cout << "Sum is " << t.n << "/" << t.d << "\n";
```

in line 37, we’d rather write

```c++
std::cout << "Sum is " << t << "\n";
```

just like we are doing it for fundamental types.

From what we have done above, you can guess that all we have to do is to overload the output operator <<. In discussing the output operator in Section 2.1.14, we have argued that the output stream passed to and returned by the output operator must be an lvalue, since the output operator modifies the stream. Having reference types at our disposal, this can easily be done: we simply pass and return the output stream (whose type is std::ostream) as a reference:
// POST: a has been written to o
std::ostream& operator<<(std::ostream& o, const rational r)
{
    return o << r.n << "/" << r.d;
}

There is no reason to stop here: for the input, we would in the same fashion like
 to replace the two input statements std::cin >> r.n; and std::cin >> r.d;
by the single statement
std::cin >> r;
(and the same for the input of s). Again, we need to pass and return the input stream
(of type std::istream) as a reference. In addition, we must pass the rational number
that we want to read as a reference, since the input operator has to modify its value.

The operator first reads the numerator from the stream, followed by a separating
character, and finally the denominator. Thus, we can read a rational number in one go
by entering for example 1/2.

// POST: r has been read from i
// PRE: i starts with a rational number of the form "n/d"
std::istream& operator>>(std::istream& i, rational& r)
{
    char c; // separating character, e.g. '/'
    return i >> r.n >> c >> r.d;
}

In contrast to operator<<, things can go wrong, e.g., if the user enters the character
sequence "A/B" when prompted for a rational number. Also, we probably don’t want to
accept 3.4 as a rational number as our input operator does. There are mechanisms to
deal with such issues, but we won’t discuss them here.

Let us conclude this section with a beautified version of Program 33 (see Program 34
below). What makes this version even nicer is the fact that the new type is used exactly
as fundamental type such as int. All the operations necessary for this are outsourced
into Program 35.

In the spirit of Section 3.1.8 on modularization, we should actually split the program
into three files: a file rational.h that contains the definition of the struct rational,
along with declarations of the overloaded operators; a file rational.cpp that contains
the definitions of these operators; and finally, a file userational2.cpp that contains
the main program. At the same time, we should put our new type rational and the
operations on it into namespace ifm in order to avoid possible name clashes. (You can
still write the expression r + s in Program 34 without mentioning the namespace in
which the operator + in question is defined. The Details of Section 4.1 explains this in
the paragraph on argument-dependent name lookup.) Exercise 136 asks you to do all
this and integrate the new rational number type into the math library that you have
built in Exercise 106 so that Program 34 below can be compiled using this library.
CHAPTER 4. COMPOUND TYPES

---

Program 34: ..//progs/lecture/userational2.cpp

---

// Program: userational2.cpp
// Add two rational numbers.
#include <iostream>
#include "rational.cpp"

int main ()
{
    // input
    std::cout << "Rational number r:\n";
    rational r;
    std::cin >> r;

    std::cout << "Rational number s:\n";
    rational s;
    std::cin >> s;

    // computation and output
    std::cout << "Sum is " << r + s << "\n";
    return 0;
}

---

Program 34: ..//progs/lecture/userational2.cpp

---

// Program: rational.cpp
// Define a type rational and operations on it

// the new type rational
struct rational {
    int n;
    int d; // INV: d != 0
};

// POST: b has been added to a; return value is the new value of a
rational& operator+= (rational& a, const rational b) {
    a.n = a.n * b.d + a.d * b.n;
    a.d *= b.d;
    return a;
}

// POST: return value is the sum of a and b
rational operator+ (const rational a, const rational b) {

4.2. REFERENCE TYPES

21 // reduce to operator +=
22 rational result = a;
23 return result += b;
24 }
25
26 // POST: a has been written to o
27 std::ostream& operator<< (std::ostream& o, const rational a)
28 {
29    return o << a.n << "/" << a.d;
30 }
31
32 // POST: a has been read from i
33 // PRE: i starts with a rational number of the form "n/d"
34 std::istream& operator>> (std::istream& i, rational& a)
35 {
36    char c; // separating character, e.g. '/'
37    return i >> a.n >> c >> a.d;
38 }

Program 35: ../progs/lecture/rational.cpp

Here is an example run of Program 34, showing that we have now achieved what we set out to do in the beginning of Section 4.1.

Rational number r:
1/2
Rational number s:
1/3
Sum is 5/6.

4.2.6 Const references

Let us come back to the addition operator for rational numbers defined in Program 35:

21 // POST: return value is the sum of a and b
22 rational operator+ (const rational a, const rational b)
23 {
24    rational result = a;
25    return result += b;
26 }

The efficiency fanatic in you might already have anticipated that we can slightly speed up this operator by using const-qualified reference types:

21 // POST: return value is the sum of a and b
22 rational operator+ (const rational& a, const rational& b)
23 {
24    rational result = a;

\begin{verbatim}
return result += b;
}
\end{verbatim}

Indeed, this version is potentially faster than the previous one, since the initialization of a formal argument is done by copying just one address, rather than two int values as in the member-wise copy that takes place under the call-by-value semantics. Even if the saving is small in this example, you can imagine that member-wise copy can be pretty expensive in structs that are more elaborate than rational; in contrast, call by reference is fast for all types, even the most complicated ones.

As usual, the const keyword makes sure that the formal arguments cannot be changed within the function body, but in addition, the const promise extends to the object referred to by the call argument, since the formal argument will become an alias for it. Under call by reference, const-correctness is thus much more important than under call by value. Here is an instructive example. Consider the unary operator- from Section 4.1.5:

\begin{verbatim}
// POST: return value is -a
rational operator- (rational a)
{
    a.n = -a.n;
    return a;
}
\end{verbatim}

Blindly speeding this up to

\begin{verbatim}
rational operator- (rational& a)
{
    a.n = -a.n;
    return a;
}
\end{verbatim}

has a drastic (and undesired) consequence: the expression \(-a\) will still have the same value as before, but it will have the additional effect of negating the value of the call argument. We have "accidentally" created a completely different operator. The fact that we have started out with a non-constant formal argument in the first place is taking revenge when we move to call by reference semantics.

Const-correctness requires that whenever an argument is passed by reference to a non-mutating function, it must have const-qualified reference type. In case of the unary minus, this would have revealed our programming error:

\begin{verbatim}
rational operator- (const rational& a)
{
    a.n = -a.n; // error: a was promised to be constant
    return a;
}
\end{verbatim}

In compiling this variant of the operator, the compiler will issue an error message, pointing out the mistake. We can then fix it by either going back to call-by-value semantics, or by introducing a result variable like in operator+ above:
rational operator- (const rational& a)
{
    rational result = a;
    result.n = -result.n;
    return result;
}

Let us discuss the const mechanism in context of references in more detail. The situation is very similar to that of const pointers, see Section 2.6.8.

A const reference is a reference for which the object referred to is constant. A const reference can for example be obtained as follows.

```c
int i = 5;
const int k = i; // j becomes a non-modifiable alias of i
```

Now, writing `j = 6` leads to an error message, since `i` and its alias `j` have been promised to be constant.

A const reference behave like (and is implemented by) a constant const pointer to the object that it refers to.

Const references of const-qualified type `const T&` can as usual be initialized with lvalues of type `T`, but also with rvalues of type `T`, or of types whose values are convertible to `T`. We can for example write

```c
const int& j = 5;
```

Under the hood, the compiler generates a temporary object that holds the value 5, and it initializes `j` with the address of the temporary object. This is useful in `operator+` that can still be called with rvalue arguments under formal arguments of const-qualified reference type `const rational&`. A formal argument of type `const T&` is therefore the all-in-one device suitable for every purpose: if the call argument is an lvalue, the initialization is very efficient (only its address needs to be copied), and otherwise, we essentially fall back to call-by-value semantics.

Despite this, there are still situations where `T` is preferable over `const T&` as argument type. If `T` is a fundamental type or a struct with small memory requirements, it does not pay off to move to `const T&`, since the saving in handling lvalue arguments is so small (or even nonexistent) that it won’t compensate for the (slightly) more costly access to the formal function argument in the function body. Indeed, call by reference adds one indirection: to look up the value of a formal function argument under call-by-reference semantics, we first have to look up its address and then look up the actual value at that address. Under call-by-value semantics, the address of the value is “hardwired” (and refers to some object on the call stack, see Section 3.2.2).

Also, it is often convenient to use the formal argument as a local variable and modify its value (see `operator-` above); for that, its type must not be a const-type.

What exactly is constant? We already had a paragraph with the same title in Section 2.6.8 and what is going on here is also the same: an expression of type `const T&` is the alias
of an object, but that object is not necessarily constant itself. The const-qualification in this case is merely a promise that the object’s value will not be modified through the alias in question. Here is an example that illustrates this point.

```c
int n = 5;
const int & i = n; // i becomes a non-modifiable alias of n
int & j = n;       // j becomes a modifiable alias of n
i = 6;            // error: n is modified through const-reference
j = 6;            // ok: n receives value 6
```

Here, we do not have a constant object, but a constant alias (namely i).

### 4.2.7 Const-types as return types.

Const-types may also appear as return types of functions, just like any other types. In that case, the const promises that the function call expression itself is constant.

It is not generally valid to replace return type \( T \) by \( const\ T & \); while this safely works for the formal argument types, it can for the return type result in syntactically correct but semantically wrong code.

As an example, let’s replace rational by \( const\ rational \& \) as the return type of \( \text{operator+} \), with the hope of realizing another speedup:

```c
const rational & operator+ (const rational & a, const rational & b) {
    rational result = a;
    return result += b;
}
```

In executing the return statement, the return value (in this case a const-reference) to be passed to the caller of the function is initialized with the expression \( \text{result} \). Now recall that the initialization of a (const-)reference from an lvalue simply makes it an alias of the lvalue. But the lvalue in question (namely \( \text{result} \)) is a local variable whose memory is freed and whose address becomes invalid when the function call terminates (see Section 2.4.3 and Section 4.2.1). The consequence is that the returned reference will be the alias of an expired object, and using this reference results in undefined behavior of the program. The issue is again the one of temporary objects, see the discussion and the Reference Guideline on 265.

Errors like this are very hard to find (and we cannot reliably count on compiler warnings here), since the program may work as intended, for example if the memory that was associated to the expired object is not immediately reused. But on another platform, the program may behave differently or even crash.

### 4.2.8 Goals

Dispositional. At this point, you should ...  
1) understand the alias concept behind reference types and the Reference Guideline;
2) understand the difference between call by value and call by reference semantics for function arguments;
3) understand const references.

Operational. In particular, you should be able to . . .

(G1) state exact pre- and postconditions for functions involving formal argument types or return types of reference and/or const-type;
(G2) write functions that modify (some of) their call arguments;
(G3) find syntactical and semantical errors in programs that are due to improper handling of reference types;
(G4) find syntactical and semantical errors in programs that are due to improper handling of const-types;
(G5) find the declarations in a given program whose types should be const-according to the Const Guideline.

4.2.9 Exercises

Exercise 136 Split program Program 35 into a header file rational.h and a source file rational.cpp. Build object code file rational.o from rational.cpp and integrate it into the library libmath.a that you have created in Exercise 106. Modify the main program Program 34 to use this library.

Exercise 137 Consider the following family of functions:

T foo (S i)
{
    return ++i;
}

with T being one of the types int, int&, and const int&, and S being one of the types int, const int, int& and const int&. This defines 12 different functions, and all of those combinations are syntactically correct.

a) Find the combinations of T and S for which the resulting function definition is semantically valid, meaning, for example, that the constness of variables and references is respected. Semantical correctness also means that the compiler will accept the code, because we have already established syntactical correctness. Explain your answer.

b) Among the combinations found in a), find the combinations of T and S for which the resulting function definition is also valid during runtime, meaning that function calls always have well-defined value and effect; explain your answer.
c) For all combinations found in b), give precise postconditions for the corresponding function foo.

Exercise 138 Write a function that swaps the values of two int-variables.

For example,

```cpp
int a = 5;
int b = 6;
// here comes your function call
std::cout << a << "\n"; // outputs 6
std::cout << b << "\n"; // outputs 5
```

Exercise 139 We want to have a function that normalizes a rational number, i.e. transforms it into the unique representation in which numerator and denominator are relatively prime, and the denominator is positive. For example,

\[
\frac{21}{-14}
\]

is normalized to

\[
\frac{-3}{2}.
\]

There are two natural versions of this function:

```cpp
// POST: r is normalized
void normalize (rational& r);
```

```cpp
// POST: return value is the normalization of r
rational normalize (const rational& r);
```

Implement one of them, and argue why you have chosen it over the other one.

Hint: you may want to use the function gcd from Section 3.2, modified for arguments of type int (how does this modification look like?).

Exercise 140 Provide a definition of the following function.

```cpp
// POST: return value indicates whether the linear equation
// a * x + b = 0 has a real solution x; if true is
// returned, the value s satisfies a * s + b = 0
bool solve (double a, double b, double& s);
```

Test your function in a program for at least the pairs (a, b) from the set

\{(2, 1), (0, 2), (0, 0), (3, -4)\}. 

Exercise 141 Find all mistakes (if any) in the following programs, and explain why these are mistakes. All programs share the following two function definitions and only differ in their main functions.

```c++
int foo (int & i) {
    return i += 2;
}

const int & bar (int &i) {
    return i += 2;
}
```

a) int main() {
    const int i = 5;
    int & j = foo (i);
}

b) int main() {
    int i = 5;
    const int & j = foo (i);
}

c) int main() {
    int i = 5;
    const int & j = bar (foo (i));
}

d) int main() {
    int i = 5;
    const int & j = foo (bar (i));
}

e) int main() {
    int i = 5;
    const int j = bar (++i);
}

Exercise 142 The C++ standard library also contains a type for computing with complex numbers. A complex number where both the real and the imaginary part are
doubles has type std::complex<double> (you need to include <complex> in order to get this type). In order to get a a complex number with real part r and imaginary part i, you can use the expression

```cpp
std::complex<double>(r, i);  // r and i are of type double
```

Otherwise, complex numbers work as expected. All the standard operators (arithmetic, relational) and mathematical functions (std::sqrt, std::abs, std::pow...) are available. The operators also work in mixed expressions where one operand is of type std::complex<double> and the other one of type double. Of course, you can also input and output complex numbers.

Here is the actual exercise. Implement the following function for solving quadratic equations over the complex numbers:

```cpp
// POST: return value is the number of distinct complex solutions
// of the quadratic equation ax^2 + bx + c = 0. If there
// are infinitely many solutions (a=b=c=0), the return
// value is -1. Otherwise, the return value is a number n
// from {0,1,2}, and the solutions are written to s1,..,sn
int solve_quadratic_equation (std::complex<double> a,
                             std::complex<double> b,
                             std::complex<double> c,
                             std::complex<double>& s1,
                             std::complex<double>& s2);
```

Test your function in a program for at least the triples (a, b, c) from the set

\{(0, 0, 0), (0, 0, 2), (0, 2, 2), (2, 2, 2), (1, 2, 1), (i, 1, 1)\}.

4.2.10 Challenges

Exercise 143 Implement the following function for solving cubic equations over the complex numbers:

```cpp
// POST: return value is the number of distinct (complex) solutions
// of the cubic equation ax^3 + bx^2 + cx + d = 0. If there
// are infinitely many solutions (a=b=c=d=0), the return
// value is -1. Otherwise, the return value is a number n
// from {0,1,2,3}, and the solutions are written to s1,..,sn
int solve_cubic_equation (std::complex<double> a,
                          std::complex<double> b,
                          std::complex<double> c,
                          std::complex<double> d,
                          std::complex<double>& s1,
                          std::complex<double>& s2,
                          std::complex<double>& s3);
```
You find a brief description of the type `std::complex<double>` in the text of Exercise 142.

Write a program that tests your function. For example, you may substitute the solutions returned by the above function into \( ax^3 + bx^2 + cx + d = 0 \) and check whether the expression indeed evaluates to (approximately) zero.

**Hint:** You find the necessary theory under the keyword Cardano's formula.
4.3 Classes

Let me tell you this in closing
I know we might seem imposing
But trust me if we ever show in your section
Believe me its for your own protection

This section introduces the concept of classes as an extension of the struct concept from Section 4.1. You will learn about data encapsulation as a distinguishing feature of classes. This feature makes type implementations more safe and flexible. You will first learn classes feature by feature for rational numbers, and then see two complete classes in connection with random number generation.

4.3.1 Encapsulation

In the previous two sections, we have defined a new struct type rational whose value range models the mathematical type \( \mathbb{Q} \) (the set of rational numbers), and we have shown how it can be equipped with some useful functionality (arithmetic and relational operators, input and output).

To motivate the transition from structs to classes in this section (and in particular the aspect of encapsulation), let us start off with a thought experiment. Suppose you have put the struct rational and all the functionality that we have developed into a nice library. In Exercise [136] you have actually done this, for the very basic version of the type rational from Program [35]. Now you have sold the library to a customer; let’s call it RAT (*Rational Thinking Inc.*). RAT is initially happy with the functionality that the library provides, and starts working with it. But then some unpleasant issues come up.

Issue 1: Initialization is cumbersome. Some code developed at RAT needs to initialize a new variable \( r \) with the rational number \( 1/2 \); for this, the programmer in charge must write

```c
rational r;  // default-initialization of r
r.n = 1;      // assignment to data member
r.d = 2;      // assignment to data member
```

The declaration `rational r` default-initializes \( r \), but the actual value of \( r \) must be provided through *two* assignments later. RAT tells you that they would prefer to initialize \( r \) from the numerator and denominator in one go, and you realize that they have a point
here. Indeed, if the programmer at RAT forgets one of the assignments, \( r \) has undefined value (and you get to handle the bug reports). If the struct is larger (consider the example of rational_vector_\( 3 \) on page 249), the problem is amplified.

**Issue 2: Invariants cannot be guaranteed.** Every legal value of the type rational must have a nonzero denominator. You have stipulated this as an invariant in Program 35 but there is no way of enforcing this invariant. It is possible for anyone to write

```cpp
rational r;
r.n = 1;
r.d = 0;
```

and thus violate the *integrity* of the type, the correctness of the internal representation.

You might argue that it would be quite stupid to write \( r.d = 0 \), and even the programmer at RAT can't be that stupid. But in RAT’s application, the values of rational numbers arise from complicated computations somewhere else in the program; these computations may result in a zero denominator simply by mistake, and in allowing value 0 to be assigned to \( r.d \), the mistake further propagates instead of being withdrawn from circulation (again, you get to handle the bug reports).

You think about how both issues could be addressed in the next release of the rational numbers library, and you come up with the following solution: As another piece of functionality on the type rational, you define a function that creates a value of type rational from two values of type int. (We have proposed such a solution already in Exercise 128 in connection with nested structs).

```cpp
// PRE: d != 0
// POST: return value is n/d
class rational
{
    rational create_rational (const int n, const int d) {
        assert (d != 0);
        rational result;
        result.n = n;
        result.d = d;
        return result;
    }
}
```

You then advise RAT to use this function whenever they want to initialize or assign to a rational number. For example,

```cpp
rational r = create_rational (1, 2);
```

would initialize \( r \) with 1/2 in one go, and at the same time make sure that the denominator is nonzero.

Such a creation function certainly makes sense for structs in general, but the two issues above don’t really go away. The reason is that this safe creation can be circumvented by not using it. In fact, your advice might not have reached the programmer at RAT, and even if it did, the programmer might be too lazy to follow it. It is therefore still possible
to write rational r; and forget about data member assignment, and it is still possible
to assign 0 to r.d. Behind this lies in fact a much larger problem, as you discover next.

Issue 3: The internal representation cannot be changed. After having used the rational
numbers library for some time, RAT approaches you with a request for a version with a
larger value range, since they have observed that intermediate values sometimes overflow.

You recall the type extended_int from Page 249 and realize that one thing you could
easily do is to change the type of numerator and denominator from int to unsigned int
and store the sign of the rational number separately as a data member of type bool, for
example like this:

```cpp
struct rational {
    unsigned int n; // absolute value of numerator
    unsigned int d; // absolute value of denominator
    bool is_negative; // sign of the rational number
};
```

It is also not too hard to rewrite the library files rational.h and rational.cpp to reflect
this change in representation.

But shortly after you have shipped the new version of your library to RAT (you have
even included the safe creation function create_rational from above in the hope to
resolve issues 1 and 2 above), you receive an angry phone call from the management of
RAT: the programmer reports that although the application code still compiles with the
new version of the library, nothing works anymore!

After taking a quick look at the application code, you suddenly realize what the
problem is: the code is cluttered up with expressions of the form expr.n and expr.d, as in

```cpp
rational r;
r.n = 1;
r.d = 2;
```

Already this particular piece of code does not work anymore: a rational number is now
represented by three data members, but the (old) application code obviously does not
initialize the (new) member of type bool. Now you regret not to have provided the
create_rational function in the first place; indeed, the statement

```cpp
rational r = create_rational (1, 2);
```

would still work, assuming that you have correctly adapted the definition of the function
create_rational to deal with the new representation. But the problem is much more
far-reaching and manifests itself in each and every occurrence of expr.n or expr.d in the
application code, since the data members have changed their meaning (they might even
have changed their names): in letting RAT access numerator and denominator through
data members that are specific to a certain representation, you are now committed to that
representation, and you can’t change it without asking RAT to change its application
code as well (which they will refuse, of course).
4.3. CLASSES

When the RAT management realizes that the new rational numbers with extended value range are useless for them, they terminate the contract with you. Disappointed as you are, you still realize that what you need to avoid such troubles in the future is encapsulation: a mechanism that hides the actual representation of the type rational from the customer, and at the same time offers the customer representation-independent ways of working with rational numbers.

In C++, encapsulation is available through the use of classes, and we start by explaining how to hide the representation of a type from the customer. In the following, the term "customer" is used in a broader sense for all programs that use the class.

4.3.2 Public and private

Here is a preliminary class version of struct rational that takes care of data hiding.

```c++
class rational {
    private:
        int n;
        int d;  // INV: d != 0
    }
```

In the same way as a struct, a class aggregates several different types into a new type, but the class keyword indicates that access restriction may occur, realized through the keywords public and private.

A data member is public if and only if its declaration appears somewhere after a public: specifier, and with no private: specifier in between. It is private otherwise. In particular, if the class definition (see Section 4.3.9 below for the precise meaning of this term) contains no public: specifier, all data members are private by default. In contrast, a struct is a class where all data members are public by default.

If a data member is private, it cannot be accessed by customers through the member access operator. If a data member is public, there are no such restrictions. Under our above definition of class rational, the following will therefore not compile:

```c++
    rational r;
    r.n = 1;    // error: n is private
    r.d = 2;    // error: d is private
    int i = r.n; // error: n is private
```

In particular, the assignment `r.d = 0` becomes impossible (which is good), but at a (too) high price: now your customer cannot do anything with a rational number, and even you cannot implement `operator+=`, say, as you used to do it in Program 35. What we are still lacking is some way of accessing the encapsulated representation. This functionality is provided by a second category of class members, namely member functions.
CHAPTER 4. COMPOUND TYPES

4.3.3 Member functions

Let us now add the missing functionality to class rational through member functions. It would seem natural to start with safe creation, but since there are specific member functions reserved for this purpose, let us first show two "general" member functions that grant safe access to the numerator and denominator of a rational number (we'll discuss below what *this and const mean here; and if you wonder why we can use n and d before they are declared: this is a special feature of class scope, explained in Section 4.3.9).

class rational {
public:
   // POST: return value is the numerator of *this
   int numerator () const
   {
      return n;
   }
   // POST: return value is the denominator of *this
   int denominator () const
   {
      return d;
   }
private:
   int n;
   int d; // INV: d != 0
};

If r is a variable of type rational, for example, the customer can then write

int n = r.numerator (); // get numerator of r
int d = r.denominator (); // get denominator of r

using the member access operator as for data members. The customer can call these two functions, since they are declared public. Access specifiers have the same meaning for member functions as for data members: a private member function cannot be called by the customer. This kind of access to the representation is flexible, since the corresponding member functions can easily be adapted to a new representation; it is also safe, since it is not possible to change the values of the data members through the functions numerator and denominator. As a general rule of thumb, all data members of a class should be private (otherwise, you encourage the customer to access the data members, with the ugly consequences mentioned in Issue 3 above).

The implicit call argument and *this. In order to call a member function, we need an expression of the class type for which we access the function, and this expression (appearing before the .) is an implicit call argument whose value may or may not be modified by the function call.
Within each member function, the lvalue *this refers to this implicit call argument and explains the appearance of *this in the postconditions of the two member functions above. It does not explain why an asterisk appears in *this, but we will get to this later.

**Const member functions.** A const keyword after the formal argument list of a member function refers to the implicit argument *this and therefore promises that the member function call does not change the value (represented by the values of the data members) of *this. We call such a member function a **const member function**.

**Member function call.** The general syntax of a member function call is

```
expr.fname( expr1, ..., exprN )
```

Here, expr is an expression of a class type for which a member function called fname is declared, expr1,..., exprN are the call arguments, and . is the member access operator. In most cases, expr is an lvalue of the class type, typically a variable.

**Access to members within member functions.** Within the body of a member function f of a class, any member (data member of member function) of the same class can be accessed without a prefix expr.; in this case, we implicitly access it for *this. In our example, the expression n in the return statement of the member function numerator refers to the data member n of *this. The call r.numerator() therefore does what we expect: it returns the numerator of the rational number r.

Within member functions, we can also access members for other expressions of the same class type through the member access operator (like a customer would do it). All accesses to class members within member functions of the same class are unrestricted, regardless of whether the member in question is public or private. The public: and private: specifiers are only relevant for the customer, but not for member functions of the class itself.

Member functions are sometimes also referred to as **methods** of the class.

**Member functions and modularization.** In the spirit of Section [3.1.8], it would be useful to source out the member function definitions, in order to allow separate compilation. This works like for ordinary functions, except that in a member function definition outside of the class definition, the function name must be qualified with the class name. In the header file rational.h we would then write only the declarations (as usual within namespace ifm):

```c
class rational {
public:
  // POST: return value is the numerator of *this
  int numerator() const;
```
CHAPTER 4. COMPOUND TYPES

// POST: return value is the denominator of *this
int denominator () const ;
private:
    int n;
    int d; // INV: d≠ 0
};

The matching definitions would then appear in the source code file rational.cpp (again within namespace ifm, and after including rational.h) as follows.

int rational::numerator () const
{
    return n;
}
int rational::denominator () const
{
    return d;
}

4.3.4 Constructors

A constructor is a special member function that provides safe initialization of class values. The name of a constructor coincides with the name of the class, and—this distinguishes constructors from other functions—it does not have a return type, and consequently no return value. A class usually has several constructors, and the compiler figures out which one is meant in a given context (using the rules of overloading resolution, see the Details of Section 4.1).

The syntax of a constructor definition for a class \( T \) is as follows.

\[
T (T_1 p\text{name}1, T_2 p\text{name}2, ..., T_N p\text{name}N) : name1 (expr1), ..., nameM (exprM)
\]

Here, \( p\text{name}1, ..., p\text{name}N \) are the formal arguments of the constructor. In the initializer

\[
: name1 (expr1), ..., nameM (exprM)
\]

\( name1, ..., nameM \) are data members, and \( expr1, ..., exprM \) are expressions of types whose values can be converted to the respective data member types. These values are used to initialize the data members, before \( block \) is executed, and in the order in which the members are declared in the class. In other words, the order in the initializer is ignored, but it is good practice to use the declaration order here as well. If a data member is not listed in the initializer, it is default-initialized. In the constructor body \( block \), we can still set or change the values of some of the data members.
For the type rational, here is a constructor that initializes a rational number from two integers.

```
// PRE: d != 0
// POST: *this is initialized with numerator / denominator
rational(const int numerator, const int denominator)
  : n(numerator), d(denominator)
{
  assert(d != 0);
}
```

To use this constructor in a variable declaration, we would for example write

```
rational r(1, 2); // initializes r with value 1/2
```

In general, the declaration

```
T x(expr1, ..., exprN)
```

defines a variable x of type T and at the same time initializes it by calling the appropriate constructor with call arguments expr1, ..., exprN.

The constructor can also be called explicitly as in

```
rational r = rational(1, 2);
```

This initializes r not directly from two integers, but from an expression of type rational that is constructed by the explicit constructor call rational(1, 2) (which is of type rational).

### 4.3.5 Default constructor

In Section 4.1.4, we have introduced the term default-initialization for the kind of initialization that takes place in declarations like

```
rational r;
```

For fundamental types, default-initialization leaves the value in question undefined, but for class types, the default constructor is automatically called to initialize the value. If present, the default constructor is the unique constructor with an empty formal argument list.

By providing a default constructor, we can thus make sure that class type values are always properly initialized. In case of the class rational (or any arithmetic type), default-initialization with value 0 seems to be the canonical choice, and here is the corresponding default constructor.

```
// POST: *this is initialized with 0
rational()
  : n(0), d(1)
{};
```
 CHAPTER 4. COMPOUND TYPES

In fact, we must provide a default constructor if we want the compiler to accept the declaration rational r. This makes class types safer than fundamental types, since it is not possible to circumvent a constructor call in declaring a variable.

The careful reader will notice that there must be an exception to this rule: Program 33 in Section 4.1 contains the declaration statement rational r; although in that program, the type rational is a struct without any constructors. This is in fact the only exception: for a class without any constructors, the default constructor is implicitly provided by the compiler, and it simply default-initializes the data members; if a data member is of class type, this in turn calls the default constructor of the corresponding class. This exception has been made so that structs (which C++ has inherited from its precursor C) fit into the class concept of C++.

4.3.6 User-defined conversions

Constructors with one argument play a special role: they are user-defined conversions. For the class rational, the constructor

```cpp
// POST: *this is initialized with value i
rational (const int i)
  : n (i), d (1)
{
}
```

is a user-defined conversion from int to rational. Under this constructor, int becomes a “type whose values can be converted to rational”. This for example means that we can provide a call argument of type int whenever a formal function argument of type rational is expected; in the implicit conversion that takes place, the converting constructor is called. With user-defined conversions, we go beyond the set of standard conversions that are built-in (like the one from int to double), but in contrast to the (sometimes incomplete) standard conversion rules stipulated by the C++ standard, we make the rules ourselves.

There are meaningful user-defined conversions that can’t be realized by constructors. For example, if we want a conversion from rational to double, we can’t add a corresponding constructor to the type double, since double is not a class type. Even conversions to some class type T might not be possible in this way: if T is not “our” type (but comes from a library, say), we cannot simply add a constructor to T. In such situations, we simply tell our type how its values should be converted to the target type. The conversion from rational to double, for example, could be done through a member function named operator double like this.

```cpp
// POST: return value is double-approximation of *this
operator double () const
{
  return double(n)/d;
}
```
In general, the member function \texttt{operator S} has implicit return type \texttt{S} and induces a user-defined conversion to the type \texttt{S} that is automatically invoked whenever this is necessary.

### 4.3.7 Member operators

All functionality of rational numbers that we have previously provided through “global” functions (\texttt{operator+, operator+=, ...}) must now be reconsidered, since directly accessing the data members is no longer possible. For example, \texttt{operator+=} as in Program 35 needs to change the value of a rational number, but there is no specific member function that allows us to do this.

The way to go is to realize \texttt{operator+} as a public member function (a \textit{member operator}), having only one formal argument (for \texttt{b}), and \texttt{*this} taking the role of \texttt{a}. This looks as follows.

```cpp
// POST: b has been added to *this; return value is
// the new value of *this
rational& operator+= (const rational& b)
{
    n = n * b.d + d * b.n;
    d *= b.d;
    return *this;
}
```

Within this member function, there is no problem in accessing the data members directly, since the access restrictions do not apply to member functions. This version of \texttt{operator+=} is as efficient as the one previously used for \texttt{struct rational}.

Prefer nonmember operators over member operators. You might argue that even \texttt{operator+} should become a member function of \texttt{class rational}, and indeed, this would probably allow a slightly more efficient implementation (without calling \texttt{operator+=}). There is one important reason to keep this operator global, though, and this has to do with user-defined conversions.

Having the conversion from \texttt{int} to \texttt{rational} that we get through the constructor

```cpp
// POST: *this is initialized with value i
rational (int i);
```

we can for example write expressions like \texttt{r + 2} or \texttt{2 + r}, where \texttt{r} is of type \texttt{rational}. In compiling this, the compiler automatically inserts a converting constructor call. Now, having \texttt{operator+} as a member would remove the second possibility of writing \texttt{2 + r}. Why? Let’s first see what happens when \texttt{r + 2} is compiled. If \texttt{operator+} is a member function, then \texttt{r + 2} “means”

```cpp
r.operator+ (2)
```
In compiling this, the compiler inserts the conversion from the call argument type int to the formal argument type rational of operator+, and everything works as expected. 2 + r, however, would mean

\[ 2. \text{operator+}(r) \]

which makes no sense whatsoever. If we write a binary operator as a member function, then the first call argument must be of the respective class type. Implicit conversions do not work here: they only adapt call arguments to formal argument types of concrete functions, but they cannot be expected to “find” the class whose operator+ has to be applied.

### 4.3.8 Nested types

There is a third category of class members, and these are *nested types*. To motivate these, let us come back to Issue 3 above, the one concerning the internal representation of rational numbers. If you think about consequently hiding the representation of a rational number from the customer, then you probably also want to hide the numerator and denominator type. As indicated in the example, these types might internally change, but in the member functions numerator and denominator, you still promise to return int-values.

A better solution would be to promise only a type with certain properties, by saying for example that the functions numerator and denominator return an integral type (Section 2.2.9). Then you can internally change from one integral type to a different one without annoying the customer. Technically, this can be done as follows.

```cpp
class rational {
public:
    // nested type for numerator and denominator
    typedef int rat_int;
    ...
    // realize all functionality in terms of rat_int
    // instead of int, e.g.
    rational (rat_int numerator, rat_int denominator); // constructor
    rat_int numerator() const; // numerator
    ...
private:
    rat_int n;
    rat_int d; // INV: d!= 0
};
```

In customer code, this can be used for example like this.

```cpp
typedef rational::rat_int rat_int;

rational r (1,2);
rat_int numerator = r.numerator(); // 1
rat_int denominator = r.denominator();  // 2
```
We already see one of the properties that the nested type `rational::rat_int` must have in order for this to work. For example, values of type `int` must be convertible to it. If you have set up everything cleanly, you can now for example replace the line

```cpp
typedef int rat_int;
```

by the lines

```cpp
typedef ifm::integer rat_int;
```

and thus immediately get exact rational numbers without any overflow issues; see also Exercise 144.

**Typedef declarations.** A *typedef declaration* introduces a new name for an existing type into its scope. It does not introduce a new type. In fact, the new name can be used synonymously with the old name in all contexts. In the above code, we see this twice: within the class `rational`, the typedef declaration introduces a nested type `rat_int`, a new name for the type `int`. In the customer code, the class's nested type (that can be accessed using the scope operator, if the nested type declaration is public) receives a new (shorter) name.

In real-life C++ code, there are nested types of nested types of nested types, ... , and typenames tend to get very long due to this. The typedef mechanism allows us to keep our code readable.

### 4.3.9 Class definitions

We now have seen the major ingredients of a class. Formally, a class definition has the form

```cpp
class T {
  class-element ... class-element
};
```

where `T` is an identifier. The sequence of `class-element`s may be empty. Each `class-element` is an access specifier (public: or private:), or a member declaration. A member declaration is a declaration statement that typically declares a member function, a data member, or a nested type. Collectively, these are called members of the class, and their names must be identifiers. A class definition introduces a new type, and this type is called a class type, as opposed to a fundamental type.

A member function definition is a declaration as well, but if the class definition does not contain the definition of a member function, this function must have a matching definition somewhere else (see Section 4.3.3). All member function definitions together form the class implementation.
Class Scope. Any member declaration of a class is said to have class scope. Its declarative region is the class definition. Class scope differs from local scope (Section 2.4.3) in one aspect. The potential scope of a member declaration is not only the part of the class definition “below” the declaration, but it spans the whole class definition, and the formal argument lists and bodies of all member function definitions. In short, a class member can be used “everywhere” in the class.

If two class definitions form disjoint declarative regions, there is no problem in using the same name for members of both classes.

4.3.10 Random numbers

We now have all the means to put together a complete and useful implementation of the type rational as a class in C++; but since we have already seen most of the necessary code in Section 4.1 and in this section, we leave this as Exercise 144 and continue here with a fresh class that has a little more entertainment in store.

Playing games on the computer would be pretty boring without some unpredictability: a chess program should not always come up with the same old moves in reaction to your same old moves, and in an action game, the enemies should not always pop up at the same time and location. In order to achieve unpredictability, the program typically uses a random number generator. This term is misleading, though, since the numbers are in reality generated according to some fixed rule, in such a way that they appear to be random. But for many purposes (including games), this is completely sufficient, and we call such numbers pseudorandom.

Linear congruential generators. A simple and widely-used technique of getting a sequence of pseudorandom numbers is the linear congruential method. Given a multiplier $a \in \mathbb{N}$, an offset $c \in \mathbb{N}$, a modulus $m \in \mathbb{N}$ and a seed $x_0 \in \mathbb{N}$, let us consider the sequence $x_1, x_2, \ldots$ of natural numbers defined by the rule

$$x_i = (ax_{i-1} + c) \mod m, \quad i > 0.$$ 

A small example is the pseudorandom number generator knuth8, defined by the following parameters.

$$a = 137, \quad c = 187, \quad m = 2^8 = 256, \quad x_0 = 0.$$ 

The sequence $x_1, x_2, \ldots$ of numbers that we get from this is

4.3. CLASSES


From here on, the sequence repeats itself (in general, the period can never be longer than m). But until this point, it appears to be pretty random (although a closer look reveals that it is not random at all; do you discover a striking sign of nonrandomness?).

In order to make the magnitude of the random numbers independent from the modulus, it is common practice to normalize the numbers so that they are real numbers in the interval $[0, 1)$.

Program 36 below contains the definition of a class random in namespace ifm for generating normalized pseudorandom numbers according to the linear congruential method. There is a constructor that allows the customer to provide the parameters $a, c, m, x_0$, and a member function operator() to get the respective next element in the sequence of the $x_i$.

```cpp
1 // Prog: random.h
2 // define a class for pseudorandom numbers.
3
4 namespace ifm {
5 // class random: definition
6 class random {
7 public:
8     // POST: *this is initialized with the linear congruential
9     // random number generator
10     //     x_i = ( a * x_{i-1} + c) mod m
11     //     with seed x0.
12     random(const unsigned int a, const unsigned int c,
13             const unsigned int m, const unsigned int x0);
14
15     // POST: return value is the next pseudorandom number
16     // in the sequence of the x_i, divided by m
17     double operator()();
18
19 private:
20     const unsigned int a_; // multiplier
21     const unsigned int c_; // offset
22     const unsigned int m_; // modulus
23     unsigned int xi_; // current sequence element
```
The function operator() has no arguments in our case (that’s why its declaration is operator()(), which admittedly looks a bit funny), and it overloads the function call operator, see Table 9 in the Appendix. In general, if expr is an expression of some class type which has the member function

\[ \text{operator}(T_1 \text{ name}_1,\ldots, T_N \text{ name}_N) \]

then expr can be used like a function: the expression

\[ \text{expr} \left( \text{expr}_1,\ldots, \text{expr}_N \right) \]

is equivalent to a call of the member function operator() with arguments expr1, ..., exprN for the expression expr. We will see such calls in Program 39 and Program 40 below.

Here is the implementation of the class random in which we see how operator() updates the value of the data member \( x_i \) to be the respective next element in the sequence of the \( x_i \).

```cpp
// Progs/lecture/random.cpp

// implement a class for pseudorandom numbers.

#include <IFM/random.h>

namespace ifm {

    // class random: implementation
    random::random(const unsigned int a, const unsigned int c,
                    const unsigned int m, const unsigned int x0)
    : a_(a), c_(c), m_(m), xi_(x0) {
    }

double random::operator()( )
    {
        // update xi according to formula, ...
        xi_ = (a_ * xi_ + c_) % m_;  
        // ... normalize it to [0,1), and return it
        return double(xi_) / m_;  
    }
}
```

Program 37: ../progs/lecture/random.cpp
Many commonly used random number generators are obtained in exactly this way. For example, the well-known generator `drand48` returns pseudorandom numbers in \([0, 1)\) according to the parameters

\[
a = 25214903917, \quad c = 11, \quad m = 2^{48},
\]

and a seed chosen by the customer. Assuming that the value range of unsigned int is \([0, \ldots, 2^{32} - 1]\), we can't realize this generator using our class `random`. Doing all the computations over the type `double` and simulating the modulo operator in a suitable way is the way to go here. It is clear that we need a large modulus to obtain a useful generator, since \(m\) is an upper bound for the number of different numbers that we can possibly get from the generator. This means that `knuth8` from above is rather a toy generator.

The game of choosing numbers. Here is a game that you could play with your friend while waiting for a delayed train. Each of you independently writes down an integer between 1 and 6. Then the numbers are compared. If they are equal, the game is a draw. If the numbers differ by one, the player with the smaller number gets CHF 2 from the one with the larger number. If the two numbers differ by two or more, the player with the larger number gets CHF 1 from the one with the smaller number. You can repeat this until the train arrives (or until one of you runs out of cash, and hopefully it's your friend).

If you think about how to play this game, it's not obvious what to do. One thing is obvious, though: you should not write down the same number in every round, since then your friend quickly learns to exploit this by writing down a number that beats your number (by design of the game, this is always possible).

You should therefore add some unpredictability to your choices. You could, for example, secretly roll a dice in every round and write down the number that it shows. But Exercise 149 reveals that your friend can exploit this as well.

You must somehow finetune your random choices, but how? In order to experiment with different distributions, you decide to define and implement a class `loaded_dice` that rolls the dice in such a way that the probability for number \(i\) to come up is equal to a presupplied value \(p_i\) (a fair dice has \(p_i = 1/6\) for all \(i \in \{1, \ldots, 6\}\)). Then you could let different loaded dices play against each other, and in this way discover suitable probabilities to use against your friend (who is by the way not studying computer science).

Program 38 shows a suitable class definition (that in turn relies on the class `random` from above, with the normalization to the interval \([0, 1)\)). We will get to the class implementation (and the meaning of the data members) in Program 39 below.

```c
1 // Prog: loaded_dice.h
2 // define a class for rolling a loaded dice.
3
4 #include "IFM/random.h>
```
namespace ifm {
    // class loaded_dice: definition
    class loaded_dice {
    public:
        // PRE: p1 + p2 + p3 + p4 + p5 <= 1
        // POST: *this is initialized to choose the number
        // i in {1,...,6} with probability pi, according
        // to the provided random number generator; here,
        // p6 = 1 - p1 - p2 - p3 - p4 - p5
        loaded_dice(double p1, double p2, double p3, double p4,
                    double p5, ifm::random& generator);

        // POST: return value is the outcome of rolling a loaded
        // dice, according to the probability distribution
        // induced by p1,...,p6
        unsigned int operator()();

    private:
        // p_upto_i is p1 + ... + pi
        const double p_upto_1;
        const double p_upto_2;
        const double p_upto_3;
        const double p_upto_4;
        const double p_upto_5;
        // the generator (we store an alias in order to allow
        // several instances to share the same generator)
        ifm::random& g;
    } // end namespace ifm
};

Program 38: ../progs/lecture/loaded_dice.h

To initialize the loaded dice, we have to provide the probabilities p1,...,p5 (p6 = 1 - \sum_{i=1}^{5} p_i), and the random number generator that is being used to actually roll the dice. Again, we overload operator() to realize the functionality of rolling the dice once. How do we implement this functionality? We partition the interval [0, 1) into 6 right-open intervals, where interval i has length p_i:

```
p1       p2       p3       p4       p5       p6
-----------
   0       p1 + p2   x       p1 + p2 + p3  1
```

Then we draw a number x at random from [0, 1), using our generator. If the number that we get were truly random, then it would end up in interval i with probability exactly
p_i. Under the assumption that our pseudorandom numbers behave like random numbers
in a suitable way, we therefore declare i as the outcome of rolling the dice if and only if
x ends up in interval i. This is the case if and only if

\[ p_1 + \ldots + p_{i-1} < x \leq p_1 + \ldots + p_i. \]

This explains the data members p_upto_1, ..., p_upto_5 (we don't need p_upto_0
(= 0) and p_upto_6 (= 1)). The constructor in Program 39 simply sets these members
from the data provided, and the implementation of operator() uses them in exactly the
way that was envisioned by the previous equation.

```cpp
#include <IFM/loaded_dice.h>

namespace ifm {

class loaded_dice : implementation
{
    loaded_dice (double p1, double p2, double p3, double p4, double p5,
                 ifm::random& generator)
    : p_upto_1 (p1),
      p_upto_2 (p_upto_1 + p2),
      p_upto_3 (p_upto_2 + p3),
      p_upto_4 (p_upto_3 + p4),
      p_upto_5 (p_upto_4 + p5),
      g (generator)
    {}

    unsigned int operator()() const
    {
        const double x = g();
        if (x <= p_upto_1 ) return 1;
        if (x <= p_upto_2 ) return 2;
        if (x <= p_upto_3 ) return 3;
        if (x <= p_upto_4 ) return 4;
        if (x <= p_upto_5 ) return 5;
        return 6;
    }
}
}
```

Program 39: ../progs/lecture/loaded_dice.cpp

Now you can compare two different loaded dices to find out which one is better in the

program of choosing numbers. Program 40 does this, assuming that you are using a loaded
dice that prefers larger numbers, and your friend uses a loaded dice that stays more in
the middle. It turns out that in this setting, you win in the long run, but not by much (CHF 0.12 on average per round). Exercise 150 challenges you to find the best loaded dice that you could possibly use in this game.

```cpp
// Prog: choosing_numbers.cpp
// let your loaded dice play against your friend's dice
// in the game of choosing numbers.

#include <iostream>
#include <IFM/loaded_dice.h>

// POST: return value is the payoff to you (possibly negative),
// given the numbers of you and your friend
int your_payoff (const unsigned int you, const unsigned int your_friend)
{
    if (you == your_friend) return 0; // draw
    if (you < your_friend) {
        if (you + 1 == your_friend) return 2; // you win 2
        return -1; // you lose 1
    } // now we have your_friend < you
    if (your_friend + 1 == you) return -2; // you lose 2
    return 1; // you win 1
}

int main () {
    // the random number generator; let us use the generator
    // ANSIC instead of the toy generator knuth8; m = 2^31;
    ifm::random ansic (1103515245u, 12345u, 2147483648u, 12345u);

    // your strategy may be to prefer larger numbers and use
    // the distribution (1/21, 2/21, 3/21, 4/21, 5/21, 6/21)
    const double p = 1.0/21.0;
    ifm::loaded_dice you (p, 2*p, 3*p, 4*p, 5*p, ansic);

    // your friend's strategy may be to stay more in the middle
    // and use the distribution (1/12, 2/12, 3/12, 3/12, 2/12, 1/12)
    const double q = 1.0/12.0;
    ifm::loaded_dice your_friend (q, 2*q, 3*q, 3*q, 2*q, ansic);

    // now simulate 1 million rounds (the train may be very late...)
    int your_total_payoff = 0;
    for (unsigned int round = 0; round < 1000000; round++) {
        your_total_payoff += your_payoff (you(), your_friend());
    }
}
```
4.3. CLASSES

4.3.11 Details

Friend functions. Sometimes, we want to grant nonmember functions access to the internal representation of a class. Typical functions for which this makes sense are the in- and output operators operator<< and operator>>. Indeed, writing out or reading into the internal representation often requires some knowledge of this representation that goes beyond what other functions need.

We cannot reasonably write operator<< and operator>> as members (why not?), but we can make these functions friends of the class. As a friend, a function has unrestricted access to the private class members. It is clear that the class must declare a function to be its friend, and not the other way around, since it's the class that has to protect its privacy, and not the function. Formally, a friend declaration is a member declaration of the form

```
friend function_declaration;
```

This declaration makes the respective function a friend of the class and grants access to all data members, whether they are public or private. For the class rational, we could rewrite the private section as follows to declare in- and output operators to be friends of the class.

```cpp
class rational {
private:
    friend std::ostream& operator<<(std::ostream& o, const rational& r);
    friend std::istream& operator>>(std::istream& i, rational& r);
    int n;
    int d; // INV: d != 0
};
```

In the definition of these operators, we can then access the numerator and denominator through .n and .d as we used to do it in Section [4.2.5]. If possible, friend declarations should be avoided, since they compromise encapsulation; but sometimes, they are useful in order to save unnecessary member functions.
4.3.12 Goals

Dispositional. At this point, you should ...

1) be able to explain the purpose of a class in C++;
2) understand the new syntactical and semantical terms associated with C++ classes, in particular access specifiers, member functions, and constructors;
3) understand the classes ifm::random and ifm::loaded_dice in detail.

Operational. In particular, you should be able to ...

(G1) find syntactical and semantical errors in a given class definition and implementation;
(G2) describe value range and functionality of a type given by a class definition;
(G3) add functionality to a given class through member functions;
(G4) write simple classes on your own;
(G5) work with and argue about pseudorandom numbers.

4.3.13 Exercises

Exercise 144 Provide a full implementation of rational numbers as a class type, and test it. The type should offer all arithmetic operators (including in- and decrement, and the arithmetic assignments), relational operators, as well as in- and output and user-defined conversions (from int and to double). As an invariant, it should hold that the internal representation is normalized (see also Exercise 139). For all the functionality you provide, decide whether it should be realized by member functions, or by nonmember functions. The class should also have a nested numerator and denominator type to achieve more flexibility, and there should be a conversion function from values of this type. Test this class with internal integral type ifm::integer!

Exercise 145 Rewrite the struct Tribool that you have developed in Exercise 129 into a class, by

a) making the data members private,

b) adding corresponding access functions,

c) adding an access function is_bool() const that returns true if and only if the value is not unknown, and

d) adding user-defined conversions from and to the type bool.
Exercise 146

a) Find all errors and violations of const-correctness in the following program. Fix them and describe the functionality of the type Clock, by providing pre- and postconditions for the member functions.

```cpp
#include <iostream>
class Clock {
    Clock(unsigned int h, unsigned int m, unsigned int s);
    void tick();
    void time(unsigned int h, unsigned int m,
               unsigned int s);
private:
    unsigned int h_;
    unsigned int m_;
    unsigned int s_;}

Clock::Clock(unsigned int & h,
             unsigned int & m,
             unsigned int & s)
    : h_(h), m_(m), s_(s) {}

void Clock::tick()
{
    h_ += (m_ += (s_ += 1) / 60) / 60;
    h_ %= 24; m_ %= 60; s_ %= 60;
}

void Clock::time(unsigned int & h,
                  unsigned int & m,
                  unsigned int & s)
{
    h = h_;
    m = m_;
    s = s_;}

int main()
{
    Clock c1(23, 59, 58);
    tick();
    unsigned int h;
    unsigned int m;
    unsigned int s;
    time(h, m, s);
    std::cout << h << ":" << m << ":" << s << ":n";
    return 0;
}
```

b) Implement an output operator for the class Clock.

Exercise 147 Write a program `random_triangle.cpp` to simulate the following random process graphically. Consider a fixed triangle T and choose an arbitrary vertex of T
as a starting point. In each step, choose as a next point the midpoint between the current point and a (uniformly) randomly selected vertex of \( t \).

The simulation at each step draws the current point into a Window. Use the window object \texttt{ifm::wio} defined in \texttt{<IFM/window>} for graphical output, and choose the triangle with vertices \((0,0)\), \((512,0)\), and \((256,512)\). Use the random number generator \texttt{ansic} from Program 40. At begin, the program should read in a seed for the random number generator and the number of simulation steps to perform. For testing purposes, let the simulation run for about 100,000 steps.

Exercise 148 Consider the generator \texttt{ansic} used in Program 40. Since the modulus is \( m = 2^{31} \), the internal computations of the generator will certainly overflow if 32 bits are used to represent unsigned \texttt{int} values. Despite this, the sequence of pseudorandom numbers computed by the generator is correct and coincides with its mathematical definition. Explain this!

Exercise 149 Find a loaded dice that beats the fair dice in the game of choosing numbers. (This is a theory exercise.)

4.3.14 Challenges

Exercise 150 What is the best loaded dice for playing the game of choosing numbers? Give its distribution! You could try to approximate the distribution experimentally, or somehow compute it. (Hint: in order to find a suitable theoretical model, search for the term “zero-sum games”, or directly go to the corresponding chapter in http://www.inf.ethz.ch/personal/gaertner/cv/lecturenotes/ra.pdf. Once you have formulated the problem as a zero-sum game, you can solve it using for example the web-interface http://banach.lse.ac.uk/form.html

Exercise 151 This is about visual cryptography. You may know the riddle of the rich Bedouin on his deathbed who wants to split his 17 camels among his three sons. Here we consider a modern version of this riddle where a secret image should be split among two people. You can for example imagine that the rich Bedouin wants to pass a treasure map to his two sons (these days, even Bedouins have less children than they used to have). Unfortunately, the two sons are at odds with each other, so that the Bedouin can’t put the map into his testament. He is afraid that the son who finds the testament first keeps the treasure for himself.

But the clever Bedouin finds a way of “splitting” the treasure map in such a way that the two sons must cooperate in order to decipher the map and find the treasure. He calls his two sons and gives each of them one image, printed on a transparency. By themselves, both images look (and really are) absolutely random, and the sons are puzzled. But then the father tells the two that in order to see the treasure map, they simply have to overlay the two transparencies!
Now how does this work? We assume for simplicity that the image to be split is black and white. The splitting happens pixel by pixel, according to the result (head or tail) of a fair coin flip. In each of the two transparencies, the original pixel is replaced by a “superpixel” of four pixels, according to the scheme in Figure 25.

Figure 25: Visual cryptography: one pixel is split into two random “superpixels” of four pixels each, depending on the outcome of a fair coin flip.

In overlaying the two transparencies, we get a black superpixel for every black pixel, and a gray superpixel for every white pixel. Thus, the overlay reveals the secret image, except that it now has double size, and white appears as gray.

Each transparency by itself does not contain any information about the secret image. Independently of the color of the original pixel, every superpixel is with the same probability in one of the two different gray states.

Now the challenge: write a program visual_crypt.cpp that splits an image in XBM format into two images (again in XBM format) according to the scheme just outlined. The XBM format is described in the text of Exercise 91 on page 175. You may use command line arguments (see the Details of Section 2.6 to specify the transparency to be generated.

Test your program by printing the resulting images on transparencies, and then overlaying them. If you cannot print on transparencies (or if you feel that you cannot afford it anymore, after a few dozen failed attempts to fix your buggy program), write a program xbm_merge.cpp that simulates the overlay of two transparencies in XBM format.
Appendix
# Appendix A

## C++ Operators

<table>
<thead>
<tr>
<th>Description</th>
<th>Operator</th>
<th>Arity</th>
<th>Prec.</th>
<th>Assoc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>scope</td>
<td>::</td>
<td>2</td>
<td>18</td>
<td>right</td>
</tr>
<tr>
<td>subscript</td>
<td>[]</td>
<td>2</td>
<td>17</td>
<td>left</td>
</tr>
<tr>
<td>function call</td>
<td>()</td>
<td>2</td>
<td>17</td>
<td>left</td>
</tr>
<tr>
<td>construction</td>
<td>type()</td>
<td>1</td>
<td>17</td>
<td>right</td>
</tr>
<tr>
<td>member access</td>
<td>.</td>
<td>2</td>
<td>17</td>
<td>left</td>
</tr>
<tr>
<td>member access</td>
<td>-&gt;</td>
<td>2</td>
<td>17</td>
<td>left</td>
</tr>
<tr>
<td>post-increment</td>
<td>++</td>
<td>1</td>
<td>17</td>
<td>left</td>
</tr>
<tr>
<td>post-decrement</td>
<td>--</td>
<td>1</td>
<td>17</td>
<td>left</td>
</tr>
<tr>
<td>dynamic cast</td>
<td>dynamic_cast&lt;&gt;</td>
<td>1</td>
<td>17</td>
<td>right</td>
</tr>
<tr>
<td>static cast</td>
<td>static_cast&lt;&gt;</td>
<td>1</td>
<td>17</td>
<td>right</td>
</tr>
<tr>
<td>reinterpret cast</td>
<td>reinterpret_cast&lt;&gt;</td>
<td>1</td>
<td>17</td>
<td>right</td>
</tr>
<tr>
<td>const cast</td>
<td>const_cast&lt;&gt;</td>
<td>1</td>
<td>17</td>
<td>right</td>
</tr>
<tr>
<td>type identification</td>
<td>typeid</td>
<td>1</td>
<td>17</td>
<td>right</td>
</tr>
<tr>
<td>pre-increment</td>
<td>++</td>
<td>1</td>
<td>16</td>
<td>right</td>
</tr>
<tr>
<td>pre-decrement</td>
<td>--</td>
<td>1</td>
<td>16</td>
<td>right</td>
</tr>
<tr>
<td>dereference</td>
<td>*</td>
<td>1</td>
<td>16</td>
<td>right</td>
</tr>
<tr>
<td>address</td>
<td>&amp;</td>
<td>1</td>
<td>16</td>
<td>right</td>
</tr>
<tr>
<td>bitwise complement</td>
<td>-</td>
<td>1</td>
<td>16</td>
<td>right</td>
</tr>
<tr>
<td>logical not</td>
<td>!</td>
<td>1</td>
<td>16</td>
<td>right</td>
</tr>
<tr>
<td>sign</td>
<td>+</td>
<td>1</td>
<td>16</td>
<td>right</td>
</tr>
<tr>
<td>sign</td>
<td>-</td>
<td>1</td>
<td>16</td>
<td>right</td>
</tr>
<tr>
<td>sizeof</td>
<td>sizeof</td>
<td>1</td>
<td>16</td>
<td>right</td>
</tr>
<tr>
<td>new</td>
<td>new</td>
<td>1</td>
<td>16</td>
<td>right</td>
</tr>
<tr>
<td>delete</td>
<td>delete</td>
<td>1</td>
<td>16</td>
<td>right</td>
</tr>
<tr>
<td>cast</td>
<td>(type)</td>
<td>1</td>
<td>16</td>
<td>right</td>
</tr>
<tr>
<td>member pointer</td>
<td>-*</td>
<td>2</td>
<td>15</td>
<td>left</td>
</tr>
<tr>
<td>member pointer</td>
<td>.*</td>
<td>2</td>
<td>15</td>
<td>left</td>
</tr>
</tbody>
</table>

...
<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
<th>Precedence</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiplication</td>
<td>*</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>division (integer)</td>
<td>/</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>modulus</td>
<td>%</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>addition</td>
<td>+</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>subtraction</td>
<td>-</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>output/left shift</td>
<td>&lt;&lt;</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>input/right shift</td>
<td>&gt;&gt;</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>less</td>
<td>&lt;</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>greater</td>
<td>&gt;</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>less equal</td>
<td>&lt;=</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>greater equal</td>
<td>&gt;=</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>equality</td>
<td>==</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>inequality</td>
<td>!=</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>bitwise and</td>
<td>&amp;</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>bitwise xor</td>
<td>^</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>bitwise or</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>logical and</td>
<td>&amp;&amp;</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>logical or</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>assignment</td>
<td>=</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>mult assignment</td>
<td>*=</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>div assignment</td>
<td>/=</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>mod assignment</td>
<td>%=</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>add assignment</td>
<td>+=</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>sub assignment</td>
<td>-=</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>rshift assignment</td>
<td>&gt;&gt;=</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>lshift assignment</td>
<td>&lt;&lt;=</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>and assignment</td>
<td>&amp;=</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>xor assignment</td>
<td>^=</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>or assignment</td>
<td></td>
<td>=</td>
<td>2</td>
</tr>
<tr>
<td>selection</td>
<td>?</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>exception</td>
<td>throw</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>sequencing</td>
<td>,</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 9: Precedences and associativities of C++ operators.
Index
Index

*this, 283
C++
   origin of name, 54
\n (line break), 152
32-bit system, 17
\0 (zero character), 152
access
   data member, 251
   member function, 282
access restrictions
   private, 281
   public, 281
access specifier
   private, 281
   public, 281
Ackermann function, 218
addition assignment operator, 54
addition operator, 50
address
   of a memory cell, 17
   of a variable, 27
address operator, 140
adjustment
   of array argument, 188
algorithm
   cache-oblivious, 19
alternate denial, 73
ANSIC
   random number generator, 295
   antivalence, 73
application program, 10
argument-dependent lookup, 255
Arithmetic Evaluation Rule 1, 47
Arithmetic Evaluation Rule 2, 47
Arithmetic Evaluation Rule 3, 48
arithmetic expression, 30
arithmetic operators, 50
arithmetic type, 47
arity, 32
array, 136
   as function argument, 188
   dimension, 155
drawbacks, 166
element, 134
fixed length, 166
incomplete type, 136
index, 137
initialization, 136
initialization from string literal, 152
initializer list, 136
multidimensional, 155
   simulation by onedimensional array, 156
of pointers, 157
out-of-bound index, 137
pointer to, 156
random access, 136
security issues, 166
size of, 138
subscript, 137
subscript operator, 137
underlying type, 136
zero-terminated, 152
array-to-pointer conversion, 142
ASCII code, 152
INDEX

assertion, 180
assignment
  member-wise, 252
  of a struct value, 251
  of reference, 264
  pointer, 140
assignment operator, 33
associative operation, 47
associativity
  left, 48
  right, 48
associativity of operator, 47
attacker, 166
automatic storage duration, 89
automaton
deterministic finite, 208

Babylonian method, 130
base
  of a floating point number system, 115
BASIC
  programming language, 188
behavior
  implementation defined, 24
  undefined, 24
  unspecified, 24
binary expansion
  of natural number, 57
  of real number, 116
binary operator, 32
binary representation
  of int value, 58
  of unsigned int value, 58
binary search, 239
binary-to-decimal conversion, 58
binomial coefficient, 237
bit, 17
bitwise operators, 74
block, 85
BODMAS, 47
body
  of do statement, 92
  of for statement, 82
  of function, 181
  of while statement, 90
bool, 70
Boole, George, 68
Boolean, 68
Boolean Evaluation Rule, 71
Boolean expression, 72
Boolean function, 68
  completeness, 69
break statement, 93
brute-force approach, 158
bubble-sort, 223
bug, 86
built-in type, 27
burst, 63
byte, 17

C++ standard, 24
cache, 19
cache-oblivious algorithm, 19
call arguments, 182
call by name, 264
call by reference, 264
call by value, 265
call stack, 213
cancellation
  in floating point computations, 124
Cardano's formula, 277
cast (functional notation), 113
cast expression, 113
central processing unit (CPU), 17
char
  literal, 151
  promotion to (unsigned) int, 151
type, 151
character, 151
  control, 152
  line break, 152
choosing numbers
game, 293
Church, Alonzo (1903–1995), 100
Church-Turing thesis, 100
class
access specifier, 289
constructor, 284
definition, 289
implementation, 289
member declaration, 289
member function, 282
member function call, 283
member operator, 287
method, 283
nested type, 288
private member, 281
public member, 281
class scope, 290
class type, 289
Collatz problem, 90
Collatz, Lothar (1910–1990), 101
command line argument, 167
command shell, 18
comment, 25
compilation, 16
    separate, 192
compiler, 16
complex number, 275
complexity
    of a problem, 219
    of an algorithm, 219
composite expression, 30
compound statement, 85
computable function, 100
computer
    main memory, 17
    memory cell, 17
    processor, 17
    von Neumann, 17
condition
    of a while statement, 90
    of an if statement, 80
conditional operator, 101
const
    member function, 283
    pointer, 143
    reference, 271
Const Guideline, 29
const pointer, 143
const reference, 271
const-correctness, 29
const-pointer type
    in function argument, 191
const-qualified type, 28
    as formal argument type of function, 182
    as return type of function, 183
    in function argument, 191
constant, 28
constant expression, 136
constant pointer, 143
constructor, 284
    default, 285
    explicit call, 285
    initializer, 284
container, 139
    iteration, 139
continue statement, 94
control character, 152
control flow, 80
    iteration, 81
    jump, 93
    linear, 80
    selection, 80
control statement, 81
control variable, 83
conversion
    array to pointer, 142
    explicit, 113
    floating point, 112
    implicit, 56
    integral, 56
    promotion, 72
    standard, 286
    user-defined, 286
correctness proof, 215
coupon collector's problem, 122
CPU, 17
Cramer's rule, 170
cryptography
    visual, 300
cryptosystem, 13

data encapsulation, 281

data member
    access for, 251
    of struct, 249
De Morgan's laws, 72
De Morgan, Augustus, 72
debugging output, 86
decimal-to-binary conversion, 57
declaration, 35
    local, 86
    of a class member, 289
    of a function, 185
    of a variable, 27
    of friend, 297
struct, 250
declaration statement, 35
declarative region, 87
default argument
    of a function, 199
default constructor, 285
default initialization
    struct, 252
    default-initialization
    by default constructor, 285
definition
    of a class, 289
    of a function, 181
    of a variable, 28
    struct, 249
delete expression, 150
denormalized number, 127
dereference operator, 141
dereferencing, 141
deterministic finite automaton, 208
DFA, 208
dimension
    (multidimensional) array, 155
directive
    include, 25
    using, 39
discriminant
    of a quadratic equation, 124
divide and conquer, 223
division assignment operator, 54
do statement, 92
    body, 92
domain
    of a function, 180
double, 110
drand48
    random number generator, 293
dynamic memory allocation, 148
dynamic programming, 158
dynamic storage duration, 148
Dynamic Storage Guideline, 151
discriminant
    of a quadratic equation, 124
divide and conquer, 223
division assignment operator, 54
do statement, 92
    body, 92
domain
    of a function, 180
double, 110
drand48
    random number generator, 293
dynamic memory allocation, 148
dynamic programming, 158
dynamic storage duration, 148
Dynamic Storage Guideline, 151
der, 15
effect
    of a function, 26
    of a statement, 35
    of an expression, 30
effect (semantical term), 27
    element
    of array, 134
    encapsulation
    of data, 281
equality
    pointer, 140
Eratosthenes' Sieve, 134
Erdős, Paul (1913–1996), 101
Euclidean algorithm, 214
Euler, Leonhard (1707 – 1783), 174
evaluation
    of an expression, 30
    order of operands, 32
    short circuit, 73
evaluation sequence, 49
Excel 2007 bug, 117
executable, 16
execution, 35
explicit conversion, 113
exponent
    of a floating point number, 115
expression, 30
arithmetic, 30
Boolean, 72
cast, 113
composite, 30
constant, 136
delete, 150
effect of, 30
evaluation of, 30
evaluation sequence, 49
function call, 182
literal, 27
lvalue, 31
mixed, 56
new, 149
of type void, 183
order of effects, 63
primary, 30
rvalue, 31
type of, 30
value of, 30
variable, 27
expression statement, 35
expression tree, 49

factoring numbers, 12
fair dice, 293
false, 70
Fibonacci numbers, 216
file, 18
finite floating point number system, 115
fixed point number, 109
float, 110
Floating Point Arithmetic Guideline 1, 121
Floating Point Arithmetic Guideline 2, 124
floating point computations
cancellation, 124
different sizes, 124
equality test, 121
relative error, 119
floating point conversions, 112
floating point number
denormalized, 127
exponent, 115
infinity, 127
mantissa, 115
NaN, 127
normalized, 115
sign, 115
significand, 115
floating point number system, 115
base, 115
largest exponent, 115
precision, 115
smallest exponent, 115
floating point type, 109
floor function, 113
for statement, 82
body, 82
init-statement, 82
iteration, 83
termination, 83
formal argument, 181
fractal, 132, 231
friend declaration, 297
function
Ackermann, 218
body, 181
call, 182
call arguments, 182
call by reference, 264
call by value, 265
const-pointer argument type, 191
const-qualified argument type, 182
const-qualified return type, 183
declaration, 185
default argument, 199
definition, 181
domain, 180
effect, 179
formal argument, 181
formal argument of reference type, 264
main, 26
mutating, 190
overloading, 253
postcondition, 179
precondition, 179
recursive, 212
recursive call, 212
return by reference, 265
return by value, 265
return type, 181
return value of reference type, 265
scope, 185
scope of formal arguments, 184
signature, 200
value, 179
void, 183
function call
qualified, 255
unqualified, 255
function call operator, 292
functional operator notation, 254
functional programming language, 26
functionality
of a struct, 251
of a type, 27
of an operator, 31
fundamental type, 27

game
choosing numbers, 293
garbage collection, 150
Gauss, Carl-Friedrich, 84
global scope, 87
global variable, 184
goto statement, 102
greatest common divisor, 214
guideline
Dynamic Storage, 151
halting problem, 84, 100
harmonic number, 122
header
file, 193
of the standard library, 26
iostream, 26
heap, 148
hexadecimal literal, 175
hiding
of name, 88
identifier, 29
IEEE compliance, 126
IEEE standard 754, 120
IEEE standard 854, 125
if statement, 80
condition, 80
if-else statement, 81
ifm::integer, 37
implementation defined behavior, 24
implicit conversion, 56
stream to bool, 154
include directive, 25
variant with angle brackets, 195
variant with quotes, 192
incomplete array type, 136
incomplete type, 250
indentation, 25
index
of array element, 137
indirection, 141
infinite loop, 83
infinite recursion, 213
infix operator notation, 254
initialization
by constructor, 285
by zero, 102
member-wise, 251
of a struct value, 251
of a variable, 36
of array, 136
of reference, 263
pointer, 140
initializer
of constructor, 284
initializer list
array, 136
input operator, 33
input stream, 33
input/output efficiency, 20
insert-sort, 223
int, 47
integer division, 52
integer division operator, 52
integral conversions, 56
integral type, 59
integrity
  of representation, 279
invariant
  of a struct, 248
iostream, 26
ISO/IEC standard 14882, 24
iteration, 81
  over a container, 139
  over an array, 139
iteration statements, 81
  equivalence of, 95
iterator, 147, 189
Josephus problem, 67
jump statements, 93
Knuth, Donald E., 13
Knuth-Morris-Pratt algorithm, 153
knuth8
  random number generator, 290
Koenig lookup, 255
layout of program, 25
least common multiple, 43
left associativity, 48
left-associative, 34
library, 194
  standard, 26
Lindenmayer system, 229
  alphabet, 228
  fractal, 231
  graphical interpretation, 230
  initial word, 229
  productions, 228
Lindenmayer, Aristide (1925–1985), 234
line break character, 152
linear congruential generator, 290
linear congruential method, 290
linear control flow, 80
linker, 193
Linux, 18
literal, 27
  bool, 70
  char, 151
  double, 111
  float, 111
  hexadecimal, 60, 175
  int, 47
  long double, 126
  octal, 60
  string, 151
  unsigned int, 56
loaded dice, 293
local declaration, 86
local scope, 87
logical parentheses, 60
logical operators, 71
logical parentheses
  leading operand, 60
  secondary operand, 60
long double, 126
long int, 62
lookup
  argument-dependent, 255
loop, 81
  infinite, 83
  progress towards termination, 83
lvalue, 31
lvalue-to-rvalue conversion, 32
Mac OS, 18
machine epsilon, 119
machine language, 15, 18
macro, 180
main function, 26
main memory, 17
Mandelbrot set, 132
mantissa
  of a floating point number, 115
mathematical induction, 215
McCarthy’s 91 Function, 236
member access
  in a struct, 251
member access operator, 251
member function, 282
  access for, 282
  and const, 283
  call, 283
  implicit call argument, 282
member operator
  of class, 287
member specification
  of a struct, 250
member-wise assignment, 252
member-wise initialization, 251
memory cell, 17
  address, 17
memory leak, 151
merge-sort, 223
  complexity, 225
Mersenne primes, 11
method
  of class, 283
minimum-sort, 219
  complexity, 220
mixed expression, 56
modularization, 191
modulus assignment operator, 54
modulus operator, 52
multidimensional array, 155
  dimension, 155
  simulation by onedimensional array, 156
multiplication assignment operator, 54
multiplication operator, 33, 50
mutating function, 190
name
  clash, 26
  hiding, 88
  of a class, 289
  of a function, 181
  of a type, 27
  of a variable, 27
  of formal argument, 181
  qualified, 26
  unqualified, 26
namespace, 26
namespace scope, 87
nested type, 288
new expression, 149
normalized floating point number, 115
null pointer, 142
null pointer value, 141
null statement, 35
numeric limits
  of floating point types, 126
  of integral types, 55
object, 30
  unnamed, 30
object code, 192
open source software, 194
operand, 31
  evaluation order, 32
operating system (OS), 18
  Linus, 18
  Mac OS, 18
  Unix, 18
  Windows, 18
operator
  addition, 50
  addition assignment, 54
  address, 140
  arithmetic, 50
  arithmetic assignment, 54
  array, 32
  assignment, 33
  associativity, 47
  binary, 32
  binding, 47
  bitwise, 74
  conditional, 101
  conversion, 286
dereference, 141
division assignment, 54
function call, 292
functional notation, 254
functionality, 31
infix notation, 254
input, 33
integer division, 52
left-associative, 34
logical, 71
modulus, 52
modulus assignment, 54
multiplication, 33, 50
multiplication assignment, 54
operator, 31
output, 34
overloading, 253
post-decrement, 53
post-increment, 53
pre-decrement, 53
pre-increment, 53
precedence, 47
relational, 71
return value, 31
sizeof, 138
subscript, 137, 146
subtraction, 50
subtraction assignment, 54
ternary, 101
unary, 32
unary minus, 52
unary plus, 52
operator token, 32
overloaded, 48
order of effects, 63
OS, 18
out-of-bound array index, 137
output operator, 34
output stream, 34
overflow
of value range, 55
overloading
argument-dependent lookup, 255
best match, 255
of functions, 253
of operators, 253
overloading resolution, 255
Panini, 241
parallel computer, 20
past-the-end pointer, 144
PEMDAS, 47
permutation, 171
perpetual calendar, 206
pipe, 78
platform, 19
point of declaration, 87, 101
pointer, 140
adding an integer, 144
arithmetic, 144
assignment, 140
comparison, 146
const, 143
constant, 143
equality, 140
initialization, 140
null, 142
null value, 141
past-the-end, 144
subscript operator, 146
subtraction, 146
to array, 156
type, 139
pointer type
underlying type, 140
porting, 16
post-decrement operator, 53
post-increment operator, 53
postcondition
of a function, 179
potential scope, 87
pre-decrement operator, 53
pre-increment operator, 53
precedence of operator, 47
precision
of a floating point number system, 115
precondition
of a function, 179
INDEX

predicate, 70
primary expression, 30
primitive recursion, 217
private
class member, 281
private:, 281
procedural programming, 188
processor, 17
production
of a Lindenmayer system, 228
program
const-correct, 29
layout, 25
valid, 23
Program Listing
callpow.cpp, 179
callpow2.cpp, 192
callpow3.cpp, 193
callpow4.cpp, 197
choosing_numbers.cpp, 297
clock.cpp, 299
collatz.cpp, 91
diff.cpp, 115
dragon.cpp, 234
eratosthenes.cpp, 135
eratosthenes2.cpp, 149
euler.cpp, 112
fahrenheit.cpp, 46
fill.cpp, 189
harmonic.cpp, 123
limits.cpp, 55
lindenmayer.cpp, 231
loaded_dice.cpp, 295
loaded_dice.h, 294
math.cpp, 196
math.h, 196
merge_sort.cpp, 228
minimum_sort.cpp, 222
perfect2.cpp, 187
pow.cpp, 191
power8.cpp, 22
power8_condensed.cpp, 25
power8_exact.cpp, 38
power8_using.cpp, 40
prime.cpp, 85
prime2.cpp, 198
random.cpp, 293
random.h, 292
rational.cpp, 269
scope.cpp, 88
shortest_path.cpp, 165
snowflake.cpp, 233
string_matching.cpp, 154
sum_n.cpp, 82
userational.cpp, 248
userational2.cpp, 268
program state, 17
programming language, 10
functional, 26
promotion, 72
bool to int, 72
char to (unsigned) int, 151
float to double, 113
pseudorandom numbers, 290
public
class member, 281
public:, 281
Pythagorean triple, 67
qualified function call, 255
qualified name, 26
quantum computer, 20
RAM, 17
random access
in array, 136
random access memory (RAM), 17
random number, 290
random number generator, 290
ANSIC, 295
drand48, 293
knuth8, 290
range
of pointers, 189
valid, 189
rational numbers, 246
recurrence relation, 225
recursion
  infinite, 213
  primitive, 217
  tail-end, 216
recursive call, 212
recursive function, 212
  correctness, 215
  termination, 215
refactoring, 97
reference, 263
  assignment, 264
  const, 271
  implementation, 263
  initialization, 263
Reference Guideline, 266
reference type, 263
relational operators, 71
relative error
  in floating point computations, 119
reserved name, 29
return by reference, 265
return by value, 265
return statement, 36 102
return type, 181
return value, 31
Reverse Polish Notation, 77
right associativity, 48
runtime error, 181
rvalue, 31
Sarrus’ rule, 171
scope
  global, 87
  local, 87
  namespace, 87
  of a declaration, 87
  of a function declaration, 185
segmentation fault, 142
selection, 80
selection statements, 81
semantical value range
  of a struct, 252
semantics, 23
sentinel, 159
separate compilation, 192
sequence point, 63
Sheffer stroke, 73
Sheffer, Henry M. (1883–1964), 73
short circuit evaluation, 73
short int, 62
shortest path problem, 158
side effect, 31
Sieve of Eratosthenes, 134
sign
  of a floating point number, 115
signature of a function, 200
signed char, 62
significand
  of a floating point number, 115
Single Modification Rule, 64
sizeof operator, 138
sorting algorithm
  bubble-sort, 223
  insert-sort, 223
  merge-sort, 223
  minimum-sort, 219
source code, 16
  availability, 194
spaghetti code, 188
special character, 151
standard conversion, 286
standard error, 34
standard input, 33
standard library, 26
  mathematical functions, 200
  std::cerr, 34
  std::cin, 33
  std::cout, 34
  std::fill, 198
  std::pow, 197
  std::sqrt, 197
standard output, 34
statement, 35
  break, 93
  compound, 85
continue, 94
control, 81
declaration, 35
do, 92
execution, 35
expression, 35
for, 82
goto, 102
if, 80
if-else, 81
iteration, 81
jump, 93
null, 35
return, 36, 102
selection, 81
switch, 99
while, 90
static storage duration, 90
static variable, 90, 102
std::cerr, 34
std::cin, 33
std::complex, 275
std::cout, 34
std::sqrt, 197
storage duration, 88
automatic, 89
dynamic, 148
static, 90
string literal, 151
string matching, 153
Knuth-Morris-Pratt algorithm, 153
obvious algorithm, 153
struct, 248
assignment, 251
data member, 249
declaration, 250
default initialization, 252
definition, 249
functionality, 251
initialization, 251
member access, 251
member specification, 250
underlying type, 249
value range, 249
semantical, 252
syntactical, 252
subscript
of array element, 137
subscript operator
array, 137
pointer, 146
subtraction assignment operator, 54
subtraction operator, 50
Sudoku, 209
swapping
of memory, 20
switch statement, 99
syntactical value range
of a struct, 252
syntax, 23
syntax error, 23
tail-end recursion, 216
temporary object, 265
reference to, 265
ternary operator, 101
topological sorting, 49
Towers of Hanoi, 240
ture, 70
Turing machine, 100
Turing, Alan (1912–1954), 100
turtle graphics, 229
two's complement, 59
type, 27
arithmetic, 47
bool, 70
built-in, 27
char, 151
class, 289
const-qualified, 28
double, 110
float, 110
floating point, 109
functionality of, 27
fundamental, 27
incomplete, 250

incomplete array, 136
int, 47
integral, 59
long double, 126
long int, 62
name of, 27
of a variable, 27
of an expression, 30
of formal argument, 181
pointer, 139
reference, 263
short int, 62
signed char, 62
underlying a struct, 249
underlying an array, 136
unsigned char, 62
unsigned int, 56
unsigned long int, 62
unsigned short int, 62
value range of, 27
void, 183
typedef, 289

Ulam spiral, 107
Ulam, Stanislaw (1909–1984), 107
unary minus operator, 52
unary operator, 32
unary plus operator, 52
undecidable problem, 100
undefined behavior, 24
underflow
 of value range, 55
underlying type
 of a pointer type, 140
Unix, 18
unnamed object, 30
unqualified function call, 255
unqualified name, 26
unsigned char, 62
unsigned int, 56
unsigned long int, 62
unsigned short int, 62
unspecified behavior, 24
user-defined conversion, 286
using directive, 39
valid program, 23
valid range, 189
value
 of a variable, 27
 of an expression, 30
value (semantical term), 27
value range
 of a struct, 249
 of a type, 27
 of type bool, 70
 of type double, 120
 of type float, 120
 of type int, 54
 of type unsigned int, 56
 overflow, 55
 semantical, 252
 syntactical, 252
 underflow, 55
variable, 27
 address of, 27
 control, 83
 global, 184
 name of, 27
 static, 90, 102
 type of, 27
 value of, 27
variable declaration, 27
variable definition, 28
vector, 166
visibility
 of name, 87
visual cryptography, 300
void, 183
void function, 183
von Neumann computer, 17
while statement, 90
 body, 90
whitespace, 155
Windows, 18
INDEX

wireframe model, 260

XBM graphics format, 175

zero-initialization, 102

zero-sum game, 300

zero-terminated array, 152