Representations of the Integers

2

A computer stores all its information by means of bits valued 0 or 1. A byte consists of eight bits. We have up to this point assumed that we could store all occurring natural numbers in the data type `int`. This, however, is in fact not the case. A variable of type `int` usually corresponds to a sequence of 4 bytes. This clearly enables us to represent no more than 2^{32} different numbers. In this chapter we will learn how integers are stored.

2.1 The b-Adic Representation of the Natural Numbers

The natural numbers are stored using the binary representation (also called the 2-adic representation). This is exactly analogous to the usual decimal representation: for example

$$\begin{aligned} 106 &= 1 \cdot 10^2 + 0 \cdot 10^1 + 6 \cdot 10^0 \\ &= 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \end{aligned}$$

with the binary representation 1101010. Instead of 10 or 2 one can choose any other natural number $b \geq 2$ as a base:

Theorem 2.1 *Let $b \in \mathbb{N}$, $b \geq 2$, and $n \in \mathbb{N}$. Then there exist uniquely defined numbers $l \in \mathbb{N}$ and $z_i \in \{0, \ldots, b-1\}$ for $i = 0, \ldots, l-1$ with $z_{l-1} \neq 0$, such that*

$$n = \sum_{i=0}^{l-1} z_i b^i \, .$$

The word $z_{l-1} \ldots z_0$ is called the ***b*-adic representation** *of n; sometimes also written as $n = (z_{l-1} \ldots z_0)_b$. The following always holds: $l - 1 = \lfloor \log_b n \rfloor$.*

S. Hougardy, J. Vygen, *Algorithmic Mathematics*,
DOI 10.1007/978-3-319-39558-6_2

Mathematical Induction
In order to prove a statement $A(n)$ for all $n \in \mathbb{N}$, it clearly suffices to show that $A(1)$ holds (initial step) and that for all $i \in \mathbb{N}$ the statement $A(i)$ implies the statement $A(i+1)$ (inductive step). In the proof of the inductive step the statement $A(i)$ is called the induction hypothesis. This method of proof is known as (mathematical) induction.
More generally, one can prove a statement $A(m)$ for all $m \in M$, where M is any set, by giving a function $f: M \to \mathbb{N}$ and showing that for every $m \in M$ the statement "$A(m)$ is false" implies that there exists an $m' \in M$ with $f(m') < f(m)$ such that $A(m')$ is also false. This is referred to as induction over f.

Proof The final statement follows immediately: if $n = \sum_{i=0}^{l-1} z_i b^i$ with $z_i \in \{0, \ldots, b-1\}$ for $i = 0, \ldots, l-1$ and $z_{l-1} \neq 0$, then $b^{l-1} \leq n \leq \sum_{i=0}^{l-1}(b-1)b^i = b^l - 1$ and thus $\lfloor \log_b n \rfloor = l - 1$.

The uniqueness of the b-adic representation now follows directly with Lemma 1.19.

The existence is proved by induction over $l(n) := 1 + \lfloor \log_b n \rfloor$; see the box **Mathematical Induction**. If $l(n) = 1$, i.e. $n \in \{1, \ldots, b-1\}$, then n has the representation $n = \sum_{i=0}^{0} z_i b^i$ with $z_0 = n$.

It remains to prove the case $n \in \mathbb{N}$ with $l(n) \geq 2$. Define $n' := \lfloor n/b \rfloor$. Then $l' := l(n') = l(n) - 1$. By the induction hypothesis n' has a representation $n' = \sum_{i=0}^{l'-1} z'_i b^i$ with $z'_i \in \{0, \ldots, b-1\}$ for $i = 0, \ldots, l'-1$ and $z'_{l'-1} \neq 0$. Define $z_i := z'_{i-1}$ for $i = 1, \ldots, l'$ and $z_0 := n \bmod b \in \{0, \ldots, b-1\}$. Then we obtain:

$$n = b\lfloor n/b \rfloor + (n \bmod b) = bn' + z_0 = b \cdot \sum_{i=0}^{l'-1} z'_i b^i + z_0 = \sum_{i=1}^{l'} z'_{i-1} b^i + z_0 = \sum_{i=0}^{l-1} z_i b^i.$$

□

If a number is given in b-adic notation as $(z_{l-1} \ldots z_0)_b$, one can find its decimal representation by using the so-called Horner scheme:

$$\sum_{i=0}^{l-1} z_i \cdot b^i = z_0 + b \cdot (z_1 + b \cdot (z_2 + \ldots + b \cdot (z_{l-2} + b \cdot z_{l-1}) \cdots))$$

saving several multiplication steps.

Conversely, the proof of Theorem 2.1 immediately yields an algorithm for obtaining the b-adic representation of a number $z \in \mathbb{N}$ for any b. Program 2.2 is a C++ implementation of this algorithm which works for $2 \leq b \leq 16$. Apart from the

binary representation ($b = 2$), the octal representation ($b = 8$) and the hexadecimal representation ($b = 16$) traditionally also play a certain role. When representing a number with base $b > 10$, one uses the letters A,B,C,. . . for the digits greater than 9.

Program 2.2 (Base Converter)

```
// baseconv.cpp (Integer Base Converter)

#include <iostream>
#include <string>
#include <limits>

const std::string hexdigits = "0123456789ABCDEF";

std::string b_ary_representation(int base, int number)
// returns the representation of "number" with base "base", assuming 2<=base<=16.
{
    if (number > 0) {
        return b_ary_representation(base, number / base) + hexdigits[number % base];
    }
    else {
        return "";
    }
}


bool get_input(int & base, int & number)           // call by reference
{
    std::cout << "This program computes the representation of a natural number"
              << " with respect to a given base.\n"
              << "Enter a base among 2,...," << hexdigits.size() << " : ";
    std::cin >> base;
    std::cout << "Enter a natural number among 1,...,"
              << std::numeric_limits<int>::max() << " : ";
    std::cin >> number;
    return (base > 1) and (base <= hexdigits.size()) and (number > 0);
}


int main()
{
    int b, n;
    if (get_input(b, n)) {
        std::cout << "The " << b << "-ary representation of " << n
                  << " is " << b_ary_representation(b, n) << ".\n";
    }
    else std::cout << "Sorry, wrong input.\n";
}
```

Program 2.2 uses the data type `string` from the C++ Standard Library. A little more background information on using this data type is given in the box **C++ in Detail** (2.1).

C++ in Detail (2.1): Strings
In order to be able to use the data type `string` from the C++ Standard Library, one first has to include the relevant part with `#include <string>`. A variable `s` of type `string` can then be defined by means of `std::string s;`. As in line 7 of Program 2.2, one can assign a value directly to a `string`-variable by entering the explicit value when defining the variable. Another useful feature is, that one can fill a `string`-variable with a given number of the same symbol. For example,

```
std::string s(10, 'A');
```

defines a `string`-variable `s` with the value `"AAAAAAAAAA"`. One can access the symbol in position `i` of a `string`-variable `s` via `s[i]`, noting, however, that the first symbol in a `string`-variable is in position 0. The expression `s1 + s2` generates a `string` which is formed by joining the `string` `s2` to the back of the `string` `s1`. The function `s.size()` returns the number of symbols contained in the `string` `s`.

In Program 2.2 we have our first example of a function which calls itself. Such functions are said to be recursive. By using them one can often make the programming code more elegant; however, one has to take care that the recursion depth, i.e. the maximum number of nested function calls, is bounded (here it is not more than 32 if the largest representable number of the data type `int` is $2^{31} - 1$). An alternative nonrecursive implementation of the function `b_ary_representation` is shown in the following excerpt of programming code:

```
1  std::string b_ary_representation(int base, int number)
2  // returns the representation of "number" with base "base", assuming 2<=base<=16.
3  {
4      std::string result = "";
5      while (number > 0) {
6          result = hexdigits[number % base] + result;
7          number = number / base;
8      }
9      return result;
10 }
```

In addition, one should note that variables are not allotted to the function `get_input` as was the case with earlier functions where values were assigned to new local variables (“call by value”), but that references are assigned to variables defined in the function `main` by means of the prefix `&` (“call by reference”). The function `get_input` thus has no variables of its own. Even more holds: the two variables `b` and `n` defined in the function `main` continue to be used in the function `get_input` with new names, `base` and `number`, respectively.

Note also, that we have defined the `string`-variable `hexdigits` in line 7 of Program 2.2 outside of every function. Thus, this variable is a **global variable** and is recognized by each of the three functions appearing in the program. In order to

prevent a value of this variable being accidentally changed, the variable has been given the prefix `const` which defines it to be a **constant**.

The representation of numbers to the base 8 or 16 can, moreover, be achieved very easily in C++ by using the stream manipulators `std::oct` and `std::hex`. Thus, for example, the instruction `std::cout << std::hex;` has the effect that all numbers in the output are given in hexadecimal notation. With `std::dec` one returns to the decimal representation of numbers.

2.2 Digression: Organization of the Main Memory

Program 2.2 serves well as an example for elucidating the organization of that part of the main computer memory which the operating system has made available for running the program. Every byte in this area has an address which is usually given in hexadecimal notation.

When a program is run, the available part of the main memory comprises four parts, often called "code", "static", "stack" and "heap" (see Fig. 2.1; further details depend on the processor and the operating system).

The part denoted code contains the program itself, given in machine code which the compiler has generated from the source code. It is partitioned into the programming code of the various functions.

In the part denoted stack, every call of a function reserves a place "on top" of the stack for the result (if it is not `void`) and for the local variables of the function (including its arguments). For any variable allotted by "call by reference", however, no new variable is formed but just the memory address of its allotted variable is stored. When running the program, the processor always keeps track of two addresses, one for the currently executed point in the code and another for the place in the stack where the range of the currently executed function begins. If a function is run recursively, then every call is assigned a new part in the stack.

In addition, two return addresses are stored for every function (except for `main`), one in the code and one in the stack, so that the program knows where and with which variables it must continue after finishing with the function, whose place in the stack is then vacated and can be reused.

There is, furthermore, some "free memory", the so-called heap, which one needs mainly when one does not know how much memory is required until running the program. For example, the data saved in a `vector` or a `string` are stored in a section of the heap. There is, however, always a corresponding normal variable in the stack, containing the storage address of the place where the section of the heap with the actual data begins. Such a variable is called a pointer.

Finally, there is a part denoted static, containing any global variables (which one should avoid if possible).

The contents of the memory are not initialized at the beginning; thus, for example, one does not know the value of a non-initialized variable. One also has no influence on the choice of storage addresses in the stack or heap. The storage

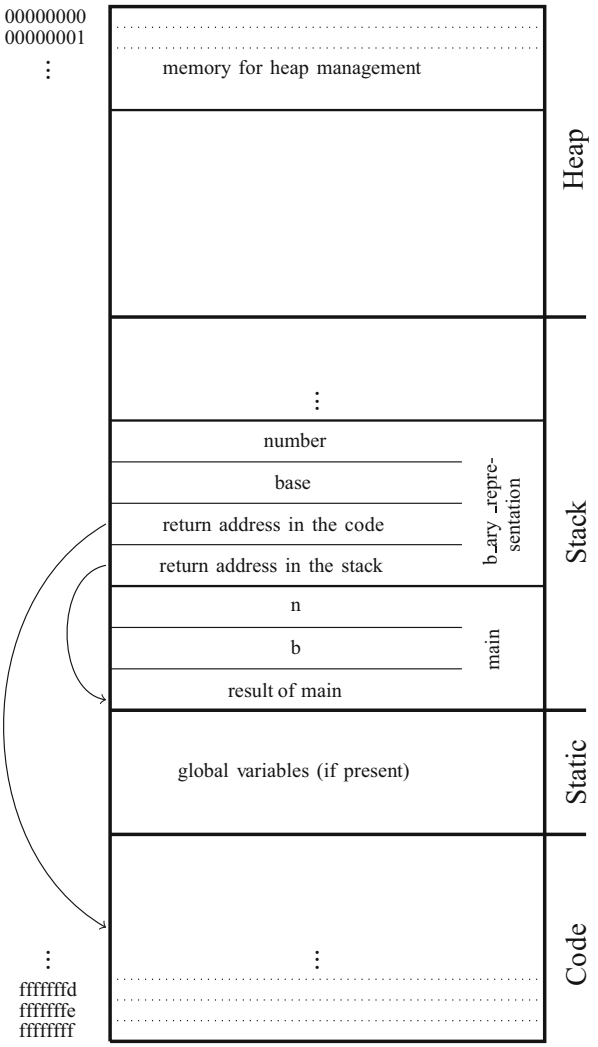

Fig. 2.1 An example of the organization of the main memory

address of a variable `x` is denoted by `&x`. Conversely, if `p` is a pointer, then `*p` denotes the content of the variable saved at the storage address `p`.

The value of a variable handed to a function via call by reference can be changed by this function (but clearly not its storage address). If one does not want to change the value, it is best to use the prefix `const` as a note for the reader of the source code and in order to prevent an inadvertent change.

2.3 The b's Complement Representation of the Integers

We are accustomed to denoting negative numbers by prefixing them with a "−" sign. In a computer this could, for example, be achieved by reserving the first bit for the sign, assigning the value 0 to the "+" sign and the value 1 to the "−" sign. This is called the sign representation.

Example 2.3 Consider the binary representation with four bits, where the first bit is reserved for the sign. The number $+5$ is given by 0101 and the number -5 as 1101. Further examples of the sign representation are:

$$\begin{array}{lll} 0 \text{ is given by } 0000 & \text{and} & -0 \text{ is given by } 1000 \\ 1 \text{ is given by } 0001 & & -1 \text{ is given by } 1001 \\ \vdots & & \vdots \\ 7 \text{ is given by } 0111 & & -7 \text{ is given by } 1111 \end{array}$$

One disadvantage of this representation is that 0 is represented in two different ways. A much more serious disadvantage is, however, that addition requires a case distinction: one cannot simply add the binary representations, as the following counterexample shows: the addition of 0010 and 1001 should yield the value 0001.

For this reason the above representation is not used in standard data types. Instead, one identifies a negative number $z \in \{-2^{l-1}, \ldots, -1\}$ with $z + 2^l$, where l is the number of bits used. This is called the 2's complement representation.

Example 2.4 With this representation using four bits we now have:

$$\begin{array}{lll} 0 \text{ is given by } 0000 & \text{and} & -8 \text{ is given by } 1000 \\ 1 \text{ is given by } 0001 & & -7 \text{ is given by } 1001 \\ \vdots & & \vdots \\ 7 \text{ is given by } 0111 & & -1 \text{ is given by } 1111 \end{array}$$

The addition of 0010 (representing the number 2) and 1001 (representing the number -7) yields 1011, which represents the number -5.

One can generalize this and define the b's complement representation of a number for any base $b \geq 2$. For this we first need to define the b's complement:

Definition 2.5 Let l and $b \geq 2$ be natural numbers and $n \in \{0, \ldots, b^l - 1\}$. Then $K_b^l(n) := (b^l - n) \bmod b^l$ is the l-digit ***b*'s complement** of n.

K_2 is also called the 2's complement and K_{10} the 10's complement.

Lemma 2.6 *Let $b, l \in \mathbb{N}$ with $b \geq 2$, and let $n = \sum_{i=0}^{l-1} z_i b^i$ with $z_i \in \{0, \ldots, b-1\}$ for $i = 0, \ldots, l-1$. Then the following statements hold:*

(i) $K_b^l(n+1) = \sum_{i=0}^{l-1}(b-1-z_i)b^i$ *if* $n \neq b^l - 1$*; furthermore,* $K_b^l(0) = 0$*;*
(ii) $K_b^l(K_b^l(n)) = n$.

Proof $K_b^l(0) = 0$ follows immediately from the definition of the b's complement. Furthermore, for $n \in \{0, \ldots, b^l - 2\}$ we have: $K_b^l(n+1) = b^l - 1 - \sum_{i=0}^{l-1} z_i b^i = \sum_{i=0}^{l-1}(b-1)b^i - \sum_{i=0}^{l-1} z_i b^i = \sum_{i=0}^{l-1}(b-1-z_i)b^i$. This completes the proof of (i).

$K_b^l(K_b^l(0)) = 0$ follows immediately from (i). For $n > 0$ we have that $K_b^l(n) = b^l - n > 0$, hence $K_b^l(K_b^l(n)) = b^l - (b^l - n) = n$. This completes the proof of (ii). □

Example 2.7 The b's complement of a positive number can be computed by means of Lemma 2.6(i): subtract 1 and then compute the difference with $b-1$ place by place, for example:

$$K_2^4((0110)_2) = (1010)_2 \qquad K_{10}^4((4809)_{10}) = (5191)_{10}$$
$$K_2^4((0001)_2) = (1111)_2 \qquad K_{10}^4((0000)_{10}) = (0000)_{10}$$

Definition 2.8 Let l and $b \geq 2$ be natural numbers and $n \in \{-\lfloor b^l/2 \rfloor, \ldots, \lceil b^l/2 \rceil - 1\}$. Then one obtains the ***l*-digit *b*'s complement representation** of n by preceding the b-adic representation of n (for $n \geq 0$) or the b-adic representation of $K_b^l(-n)$ (for $n < 0$) with as many zeros as necessary to yield an l-digit number.

The main advantage of the b's complement representation is that it enables easy computing without having to deal with different cases, as is shown in the following theorem.

Theorem 2.9 *Let l and $b \geq 2$ be natural numbers and $Z := \{-\lfloor b^l/2 \rfloor, \ldots, \lceil b^l/2 \rceil - 1\}$. Consider the function $f: Z \to \{0, \ldots, b^l - 1\}$ defined by $z \mapsto \begin{cases} z & \text{for } z \geq 0 \\ K_b^l(-z) & \text{for } z < 0 \end{cases}$*
Then f is bijective and for $x, y \in Z$ the following two statements hold:

(a) *If $x + y \in Z$, then $f(x+y) = (f(x) + f(y)) \bmod b^l$;*
(b) *If $x \cdot y \in Z$, then $f(x \cdot y) = (f(x) \cdot f(y)) \bmod b^l$.*

Proof For $z \in Z$ we have $f(z) = (z + b^l) \bmod b^l$. As $|Z| = b^l$, it follows that f is bijective. Let $x, y \in Z$ and $p, q \in \{0, 1\}$ with $f(x) = x + pb^l$ and $f(y) = y + qb^l$. Then the following two statements hold:

(a) $f(x+y) = (x + y + b^l) \bmod b^l = ((x + pb^l) + (y + qb^l)) \bmod b^l = (f(x) + f(y)) \bmod b^l$.

(b) $f(x{\cdot}y) = (x{\cdot}y{+}b^l) \bmod b^l = ((x{+}pb^l){\cdot}((y{+}qb^l)) \bmod b^l = (f(x){\cdot}f(y)) \bmod b^l$. □

The 2's complement representation is used for storing integers in the computer. In this representation the first bit equals 1 if and only if the represented number is negative. The number l of bits used is nearly always a power of 2 and a multiple of 8. For the data type `int`, for example, usually $l = 32$ (see below), permitting the representation of all numbers in the range $\{-2^{31}, \ldots, 2^{31} - 1\}$.

Partitions and Equivalence Relations
Let S be a set. A **partition** of S is a set of nonempty, pairwise disjoint subsets of S whose union is S. The set $\{1, 2, 3\}$, for example, has five different partitions.
A relation $R \subseteq S \times S$ is called an **equivalence relation** (on S) if the following three statements hold for all $a, b, c \in S$:

- $(a, a) \in R$ (reflexivity);
- $(a, b) \in R \Rightarrow (b, a) \in R$ (symmetry);
- $((a, b) \in R \wedge (b, c) \in R) \Rightarrow (a, c) \in R$ (transitivity).

For every equivalence relation R on S the set $\{\{s \in S : (a, s) \in R\} : a \in S\}$ is a partition of S; its elements are called the **equivalence classes** of R. Conversely, every partition $\mathcal{P}$ of S induces an equivalence relation on S defined by the relation $R := \{(a, b) \in S \times S : \exists P \in \mathcal{P}$ with $a, b \in P\}$.
For example, $=$ is an equivalence relation on $\mathbb{R}$; each of its equivalence classes contains just one element. On $\mathbb{Z}$ the relation $R_k := \{(x, y) \in \mathbb{Z} \times \mathbb{Z} :$ k divides $|x{-}y|\}$ defines a different equivalence relation for every $k \in \mathbb{N}$, with exactly k (infinite) equivalence classes. The set of equivalence classes of R_k is often denoted by $\mathbb{Z}/k\mathbb{Z}$ and is then called the ring of residue classes of $\mathbb{Z}$ mod k.

If one ignores numbers lying outside the stipulated range Z (or always calculates mod b^l), one can thus use the b's complement representation just like the b-adic representation. In both cases one in fact uses the elements of the residue class ring $\mathbb{Z}/b^l\mathbb{Z}$; see the box **Partitions and Equivalence Relations**. Just the representatives of some of the equivalence classes differ in the two representations.

Subtraction can be reduced to addition, and comparisons by prior subtraction to testing the first bit.

The b's complement representation was already used in the seventeenth century by designers of mechanical calculating machines. One of the largest collections worldwide of such machines is housed and can be viewed in the Arithmeum in Bonn.

Numbers lying outside the stipulated range are not normally intercepted automatically. One therefore has to watch this when programming and, if necessary, check by prior testing whether a (usually unwanted) overflow can occur.

The following table lists the most important C++ data types for integers. The numbers are represented in the 2's complement representation, so the least or greatest representable number is given by the number of bits used.

Data type	Number of bits with `gcc 6.1.0` Windows 10	Number of bits with `gcc 6.1.0` CentOS 7
`short`	16	16
`int`	32	32
`long`	32	64
`long long`	64	64

The number of bytes required by a data type or a variable can be ascertained by means of `sizeof (Datatype)` or `sizeof Variable` respectively. The result is of type `size_t` which can store nonnegative integers of sufficient size. So there is no reason to assume blindly that, for example, an `int` really has 4 bytes.

We have seen that the standard data types only allow us to store integers within a certain interval. This is often problematic; in a Collatz sequence, for example, numbers larger than 2^{64} can appear, even when the input is much smaller (cf. Program 1.25). If one does not intercept this, the computation usually continues mod 2^l, where l is the number of bits of the data type.

2.4 Rational Numbers

One can, of course, store rational numbers by allotting separate variables to numerator and denominator. Here it makes sense to define a new data type; called a **class** in C++. Classes constitute the most important construct in C++, so we will discuss them in some detail here.

A class enables one to store data in variables and provides functions that can operate on this data. Furthermore, types and constants can be defined. Of decisive importance is the clear distinction between the internal data management and the interface with the exterior. Consider Program 2.10, which consists of two files. In the file `fraction.h` a class called `Fraction` has been defined, which is suitable for storing rational numbers. Program `harmonic.cpp` is an example of how this class can be used: in fact very much like a standard data type.

The members of a class are either `public` or `private`. The part called `public` comprises the interface with the exterior. The part called `private` should contain the data (here the numerators and denominators); possibly also functions that are only needed as subroutines of the `public` functions and should not

themselves be visible from the outside. Everything that is `private` is not directly accessible from the outside. This helps to prevent programming errors.

A class can contain four kinds of functions:

- Constructors for generating an object in the class, very much like the way a variable of a standard data type is declared (and possibly also initialized). A constructor always has the same name as the class itself. In `fraction.h` the (in this case single) constructor is defined in lines 9–12. If no constructor is defined, the compiler defines a standard constructor which initializes the objects belonging to the class.
- A destructor, which is called when the lifetime of an object in the class terminates, just like variables of a function are vacated after terminating the function. As no destructor has been defined explicitly here (it would be called `~Fraction`), this role is taken on by a standard destructor generated by the compiler.
- Further functions which always operate on an existing object in the class. They can also only be called in connection with an object. They can either change this object (as for example `reciprocal()`) or not (as for example `numerator()`); in the second case one should write `const` behind the declaration. Otherwise they behave like normal functions.
- Operators, predefined for certain standard data types, can also be defined for a new class (as shown here for < and +). Operators behave just like functions; except for their special name and their more comfortable call (see for example line 16 of `harmonic.cpp`) there are no differences.

As with other functions, operators can, apart from the object in the class with which they are associated, also be assigned another object in the same class as a parameter and can, as a result, also generate a new object in the class. Their data and functions can also be accessed with "`.`". The addition shown here is an example of this.

Constructors and destructors can be called explicitly or implicitly in different ways. Lines 14 and 16 of `harmonic.cpp` show two explicit calls of the constructor, another one takes place in lines 44–46 of `fraction.h`.

Program 2.10 (Rational and Harmonic Numbers)

```
1  // fraction.h (Class Fraction)
2
3  #include <stdexcept>
4
5  class Fraction {
6  public:
7      using inttype = long long;
8
9      Fraction(inttype n, inttype d): _numerator(n), _denominator(d)
10     {   // constructor
11         if (d < 1) error_zero();
12     }
13
14     inttype numerator() const                    // return numerator
```

```
    {
        return _numerator;
    }

    inttype denominator() const                 // return denominator
    {
        return _denominator;
    }

    void reciprocal()                           // replaces a/b by b/a
    {
        if (numerator() == 0) {
            error_zero();
        } else {
            std::swap(_numerator, _denominator);
            if (denominator() < 0) {
                _numerator   = -_numerator;
                _denominator = -_denominator;
            }
        }
    }

    bool operator<(const Fraction & x) const     // comparison
    {
        return numerator() * x.denominator() < x.numerator() * denominator();
    }

    Fraction operator+(const Fraction & x) const // addition
    {
        return Fraction(numerator() * x.denominator() +
                        x.numerator() * denominator(),
                        denominator() * x.denominator());
    }

    // further operations may be added here

private:
    inttype _numerator;
    inttype _denominator;

    void error_zero()
    {
        throw std::runtime_error("Denominator < 1 not allowed in Fraction.");
    }
};
```

```
// harmonic.cpp (Harmonic Numbers)

#include <iostream>
#include <stdexcept>
#include "fraction.h"

int main()
{
try {
    std::cout << "This program computes H(n)=1/1+1/2+1/3+...+1/n.\n"
              << "Enter an integer n: ";
    Fraction::inttype n;
    std::cin >> n;
    Fraction sum(0,1);
    for (Fraction::inttype i = 1; i <= n; ++i) {
        sum = sum + Fraction(1, i);
    }
    std::cout << "H(" << n << ") = "
              << sum.numerator() << "/" << sum.denominator() << " = "
              << static_cast<double>(sum.numerator()) / sum.denominator() << "\n";
}
```

```
22 catch(std::runtime_error e) {
23     std::cout << "RUNTIME ERROR: " << e.what() << "\n";
24     return 1;
25 }
26 }
```

The output operator << can also be redefined. We could then replace the output in line 19 of `harmonic.cpp` by `<< sum`. Note, however, that in this case the first operand of << is of type `ostream`. So the new operator cannot be defined within the class `Fraction`, it has to be defined as a normal function outside `Fraction`. One could, for example, write the following code:

```
1 // output operator for class Fraction
2
3 std::ostream & operator<<(std::ostream & os, const Fraction & f)
4 {
5     os << f.numerator() << "/" << f.denominator();
6     return os;
7 }
```

Program 2.10 also yields our first example of how to deal with errors occurring while a program is running. Here one uses so-called `exceptions` which are explained in more detail in the box **C++ in Detail (2.2)**. In line 29 of `fraction.h` we have used the `swap` function which interchanges the contents of the two allotted variables.

C++ in Detail (2.2): Handling Errors with `try-catch`

During the running time of a program numerous errors can occur, for which it is not immediately clear how to deal with them at the place in the code where the error was found. Typical errors occurring while the program is running are, for example, division by 0 or the range overflow of an `int` type. The idea behind error handling in C++ is, that the part of the programming code where the error was identified reports it to the place in the code where one knows how to deal with it. The various methods of handling errors are defined in the C++ Standard Library under the header `stdexcept`. This header must therefore be included with `#include <stdexcept>`. If a runtime error is noticed, it can be referred to the calling part of the programming code with

`throw std::runtime_error (`*`error_message`*`);`

Here *`error_message`* is a `string` describing the relevant error. Line 57 of `fraction.h` provides an example. A runtime error referenced by `throw` can be suitably treated with a `try-catch` construct. This takes the general form:

`try {`*`subprogram`*`} catch (std::runtime_error e) {`*`error handling`*`}`

(continued)

An example is found in lines 9–25 of Program `harmonic.cpp`. If an error which is referenced by `throw` occurs in *`subprogram`*, then *`subprogram`* is terminated and a test is run as to whether there is a `catch` instruction matching the error type and if so, the part *`error handling`* is executed. Alongside `runtime_error` several other error types are defined in `stdexcept`. They all have one property in common, namely that one can call up the error message text stored in the `throw` instruction by means of `.what()`.

As soon as a `throw` instruction has been executed, the program continues with the error handling part of a matching `catch` instruction. If no matching `catch` instruction is available, the program returns recursively to the calling functions until a matching `catch` instruction is found. If none of the calling functions contains a matching `catch` instruction, the program terminates.

Finally, line 20 of `harmonic.cpp` shows the conversion of an integer into the data type `double`. One can in general convert an expression into another data type by means of `static_cast<`*`Datatype`*`> (`*`Expression`*`)`.

The class `Fraction` has not yet been properly implemented. At least, zero denominators are intercepted. But until now, nothing happened in the case of range overflows of `inttype`. It is, however, exactly these that can quickly occur; Program `harmonic.cpp`, for example, only works correctly for $n \leq 20$.

Of course one could, and should, always reduce all occurring fractions. This will be dealt with in the next chapter. But even this does not always prevent overflows. One could intercept these with `throw`, but it would of course be better if numerators and denominators can become arbitrarily large.

2.5 Arbitrarily Large Integers

We will now investigate the structure of a class that is able to represent arbitrarily large integers. Strictly speaking, we would have to make available all the standard operations of arithmetic as well as the comparison operators, as with standard data types like `int`. For simplicity we will consider only the operators `+`, `+=`, `<`, as well as one output function. The `+=` operator enables one to write expressions of the form `a = a + b;` more briefly as `a += b;`.

For the sake of clarity Program 2.11 is divided into three parts. The files `largeint.h` and `largeint.cpp` define a new type (class) called `LargeInt`.

The file `largeint.h` contains the declaration of the class (consisting of the parts `public` and `private`). The file `largeint.cpp` contains the implementation of all the functions belonging to the class. The division into a header file and an implementation file is quite usual and also practical when the classes are large.

The class `LargeInt` stores an arbitrarily large natural number in a `vector` called `_v`, where every place of the `vector` corresponds to a decimal place of the number. In addition, there is a constant `string` called `digits`, which is required for the numerical representation change. It is, however, not necessary for every object of type `LargeInt` to contain a copy of this `string`. This is the reason for the prefix `static`, which ensures that `digits` is stored only once for all objects in the class. The initialization of a `static` variable in a class must take place outside the class. In our case this happens in line 5 of `largeint.cpp`. Users of the class need not, however, know all these details, as they are all in the `private` part. Only the following functions are visible from the outside:

- One can form a new variable of type `LargeInt` by means of the constructor. Its argument is the number to be stored. A constructor automatically calls the constructors for the variables contained in the class; in this instance the constructor of `vector`.
- The function `decimal` outputs the decimal representation of the number in the form of a `string`.
- The comparison operator < is implemented and it tests whether or not the stored value is less than the one in the argument.
- The operator += is implemented and is used for implementing the operator +. This enables one to add numbers of type `LargeInt`.

Program 2.11 (Arbitrarily Large Integers)

```
1  // largeint.h (Declaration of Class LargeInt)
2
3  #include <vector>
4  #include <string>
5
6
7  class LargeInt {
8  public:
9      using inputtype = long long;
10
11     LargeInt(inputtype);                          // constructor
12     std::string decimal() const;                  // decimal representation
13     bool operator<(const LargeInt &) const;       // comparison
14     LargeInt operator+(const LargeInt &) const;   // addition
15     const LargeInt & operator+=(const LargeInt &); // addition
16
17 private:
18     std::vector<short> _v;  // store single digits, last digit in _v[0]
19     static const std::string digits;
20 };
```

```
1  // largeint.cpp (Implementation of Class LargeInt)
2
3  #include "largeint.h"
4
5  const std::string LargeInt::digits = "0123456789";
6
7  LargeInt::LargeInt(inputtype i)   // constructor, calls constructor of vector
8  {
9     do {
10        _v.push_back(i % 10);
11        i /= 10;
```

```
    } while (i > 0);
}


std::string LargeInt::decimal() const   // returns decimal representation
{
    std::string s("");
    for (auto i : _v) {       // range for statement: i runs over all
        s = digits[i] + s;    // elements of _v
    }
    return s;
}


bool LargeInt::operator<(const LargeInt & arg) const   // checks if < arg
{
    if (_v.size() == arg._v.size()) {
      auto it2 = arg._v.rbegin();
      for (auto it1 = _v.rbegin(); it1 != _v.rend(); ++it1, ++it2) {
        if (*it1 < *it2) return true;
        if (*it1 > *it2) return false;
      }
      return false;
    }
    return _v.size() < arg._v.size();
}


LargeInt LargeInt::operator+(const LargeInt & arg) const  // addition
{
    LargeInt result(*this);
    result += arg;
    return result;
}


const LargeInt & LargeInt::operator+=(const LargeInt & arg)   // addition
{
    if (arg._v.size() > _v.size()) {
        _v.resize(arg._v.size(), 0);
    }
    auto it1 = _v.begin();
    for (auto it2 = arg._v.begin(); it2 != arg._v.end(); ++it2, ++it1) {
        *it1 += *it2;
    }
    short carry = 0;
    for (auto & i : _v) {
        i += carry;
        carry = i / 10;
        i %= 10;
    }
    if (carry != 0) _v.push_back(carry);
    return *this;
}
```

```
// factorial.cpp (Computing n! by Addition)

#include <iostream>
#include <limits>
#include "largeint.h"


LargeInt factorial(LargeInt::inputtype n)
// computes n! warning, slow: runtime O(n^3 log n)
{
    LargeInt result(1);
    for (LargeInt::inputtype i = 2; i <= n; ++i) {
```

```
13          LargeInt sum(0);
14          for (LargeInt::inputtype j = 1; j <= i; ++j) {
15              sum += result;
16          }
17          result = sum;
18      }
19      return result;
20  }
21
22
23  int main()
24  {
25      LargeInt::inputtype n;
26      std::cout << "This program computes n!, for a natural number n up to "
27                << std::numeric_limits<LargeInt::inputtype>::max() << "\n"
28                << "Enter a natural number n: ";
29      std::cin >> n;
30      std::cout << n << "! = " << factorial(n).decimal() << "\n";
31  }
```

The constructor for `LargeInt` attaches elements stepwise to the end of the `vector _v` by means of `_v.push_back`. In lines 50 and 51 of `largeint.cpp` we see two new `vector` functions. The number of elements contained in a `vector` is returned by the function `size()`. One can change the size of a `vector` with `resize`. The first argument of this function is the number of elements and the optional second argument is a set of initial values for the new elements introduced by `resize`.

Addition as well as comparison of two `LongInts` necessitate running through all the elements of the relevant `vectors`. One could do this by accessing the various elements of `vector` by their indices. A more universal method, however, is based on the use of so-called iterators which are explained in more detail in the box **C++ in Detail (2.3)**.

C++ in Detail (2.3): Iterators and the Range-Based `for`

The C++ Standard Library provides several more so-called container types apart from the abstract data type `vector`. One can access the elements of a `vector` with the index operator `[]`. But this is not necessarily the case for other container types. In order to write code which is as independent of the used container type as possible, one can access the elements of a container by means of iterators. The function `begin()` yields an iterator referencing the first element of the container, the function `end()` yields an iterator referencing the next place after the last element of the container. In line 53 of `largeint.cpp` we have defined an iterator called `it1`, to which the value `_v.begin()` is assigned. Here we have used the type `auto`. This keyword orders the compiler to independently find the matching type by means of the assigned value. One can increase and decrease an iterator with `++` and `--`.

(continued)

This causes it to reference the next or the previous element, respectively. If one uses the functions `rbegin()` and `rend()` instead of `begin` and `end`, one can run through all the elements of the containers in reverse order by means of ++. We have used this in line 30 of `largeint.cpp`, for example. With `*iter` one obtains the current element referenced by the iterator `iter`.

It often happens that one wants to run through all the elements of a container. For this purpose the C++ Standard Library provides a special version of the `for` instruction, called the range-based `for` instruction. We have used this in lines 19–21 of `largeint.cpp`, for example. The variable `i` defined there runs through all the elements of vector `_v`. Alternatively we could also have written the loop in lines 19–21 in the following way:

```
for (auto it = _v.begin(); it != _v.end(); ++it) {
    s = digits[*it] + s;
}
```

Note the following: in a range-based `for` loop one runs through all the elements of the container, hence one does not use the dereferencing operator * as one does with iterators. If one also wants to be able to change the elements that the variable runs through, then the variable must be defined by means of & as a reference to these elements. We have used this in the `for` loop in lines 58–62 of `largeint.cpp`, for example.

The addition algorithm for two `LargeInt` numbers is initiated with the += operator in lines 48–65 of `largeint.cpp`. This operator is then used in lines 40–45 for initiating the + operator. This is done by first storing in line 42 a copy of the object to the left of the + operator. In order to do this, we must reference the object with which the + operator was called. We do this by using `this`, which contains the address of the object with which the corresponding function was called.

When there are several different variables of type `LargeInt`, as for example in the function `main` in the file `factorial.cpp`, these clearly occupy different areas of the memory. During every run through the `for` loop, a new `LargeInt` variable called `sum` is created in line 12 of `factorial.cpp`. At the end of the `for` loop this variable is deleted again by calling the destructor of `LargeInt`.

In our case all this takes place automatically: as we did not define our own destructor, the standard destructor is called (automatically); the latter then simply calls the destructor of `vector`. As `vector` puts data on top of the heap, this class has a more complicated destructor for vacating the relevant place in the memory.

The implementation of `factorial.cpp` is not very efficient and could clearly be simplified and improved by using multiplication. This, as well as other basic operations of arithmetic, can be supplemented in the class `LargeInt`. At the moment the running time is dominated by $\Theta(n^2)$ additions in each of which the larger summand has $\Theta(n \log n)$ digits, i.e. we have $\Theta(n^3 \log n)$ elementary operations. If one implements multiplication, one can compute $n!$ with $\Theta(n)$

multiplications in each of which one factor has at most $n \log n$ digits and the other at most $\log n$ digits; this yields a total running time of $O(n^2 \log^2 n)$ (if multiplication is implemented with the school method).

Such a new class is very easy to use. If, for example, one wants to implement the Collatz sequence (Program 1.25) for arbitrarily large numbers, one just has to replace line 5 of that program by `using myint = LargeInt;` and, of course, insert `#include "largeint.h"` as well. Also our class `Fraction` can work with all fractions by exchanging just one line (replace line 7 in `fraction.h` by `using inttype = LargeInt;`). Of course, we must first implement the missing operations, in particular multiplication.

http://www.springer.com/978-3-319-39557-9

Algorithmic Mathematics

Hougardy, S.; Vygen, J.

2016, XIII, 163 p. 41 illus., 40 illus. in color., Hardcover
ISBN: 978-3-319-39557-9