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.

Indiana House Bill No. 246, defining $\frac{2\sqrt{2}}{\pi} = \frac{7}{8}$, $\sqrt{2} = \frac{10}{7}$, *and* $\frac{1}{\pi} = 5/16$ *(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.C` in Section 2.2, we make mistakes. For example, 28 degrees Celsius are 82.4 degrees Fahrenheit, but not 82 as output by `fahrenheit.C`. 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.C` 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 platform, 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

$$significand \cdot 10^{exponent}.$$

For example,

$$\begin{aligned} 82.4 &= 824 \cdot 10^{-1}, \\ 0.0824 &= 824 \cdot 10^{-4}. \end{aligned}$$

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 10 does exactly this.

```
1  // Program: euler.C
2  // Approximate Euler's constant e.
3
4  #include <iostream>
5
6  int main ()
7  {
8    // values for term i, initialized for i = 0
9    float t = 1.0f;    // 1/i!
10   float e = 1.0f;    // i-th approximation of e
11
12   std::cout << "Approximating the Euler constant...\n";
13   // steps 1,...,n
14   for (unsigned int i = 1; i < 10; ++i) {
```



```
15     e += t /= i;     // compact form of t = t / i; e = e + t
16     std::cout << "Value after term " << i << ": " << e << "\n";
17   }
18
19   return 0;
20 }
```

Program 10: *progs/euler.C*

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(poten
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 42), 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. In particular, if the original integer value is in the value range of the relevant floating point number type, the value remains unchanged. If there are two nearest values, it is implementation-defined which one is chosen.

This in particular explains why the change of `int celsius` to `float celsius` in the program `fahrenheit.C` 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.C`, 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`.

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, any 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

$$\texttt{bool} \prec \texttt{int} \prec \texttt{unsigned int} \prec \texttt{float} \prec \texttt{double}.$$

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

```
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 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* $\lfloor x \rfloor$ 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 `int i = -1.6f;` To obtain the `int` value resulting from the `float` value -1.6, we can simply write the expression `int(-1.6f)`.

The general syntax of an explicit conversion, also called a *cast expression*, is

T (*expr*)

where *T* is a type, and *expr* is an expression. The cast expression is valid if and only if the corresponding conversion of *expr* to the type *T* (as in `T x =` *expr*) is defined.

For certain "complicated" type names *T*, it is necessary to parenthesize *T*, like in the cast expression `(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

```
std::numeric_limits<double>::max()
```

you might for example get the output `1.79769e+308`. Recall that this means $1.79769 \cdot 10^{308}$, 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 11 performs this task.

```
// Program: diff.C
// Check subtraction of two floating point numbers

#include <iostream>

int main()
{
  // Input
  float n1;
  std::cout << "First number     =? ";
  std::cin >> n1;

  float n2;
  std::cout << "Second number    =? ";
  std::cin >> n2;

  float d;
  std::cout << "Their difference =? ";
  std::cin >> d;

  // Computation and output
  std::cout << "Computed difference - input difference = "
            << n1 - n2 - d << ".\n";
  return 0;
}
```

Program 11: *progs/diff.C*

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 = 2.23517e-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*), $e_{\min} \in \mathbb{Z}$ (the *smallest exponent*) and $e_{\max} \in \mathbb{Z}$ (the *largest exponent*).

The set $\mathcal{F}(\beta, p, e_{\min}, e_{\max})$ 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 $e \in \{e_{\min}, \ldots, e_{\max}\}$.

The number s is the *sign*, the sequence $d_0 d_1 \ldots d_{p-1}$ is called the *significand*[17], and the number e is the *exponent*.

We typically 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 $\mathcal{F}^*(\beta, p, e_{\min}, e_{\max})$ 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 $\beta^{e_{\min}}$ (see also Exercise 46).

[17]an older equivalent term is *mantissa*

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[18]

$$(\beta-1).(\beta-1)\ldots(\beta-1) \cdot \beta^e = \sum_{i=0}^{p-1} (\beta-1)\beta^{-i} \cdot \beta^e = \left(1 - \left(\frac{1}{\beta}\right)^p\right) \beta^{e+1} < \beta^{e+1}.$$

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 11.

Computing the floating point representation. In order to convert a given positive[19] 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 i, $\underline{i}$ and $\overline{i}$, for which b_i is nonzero (note that $\underline{i}$ may be $-\infty$, but $\overline{i}$ is finite since x is finite). The number $\overline{i} - \underline{i} + 1 \in \mathbb{N} \cup \{\infty\}$ is the number of *significant digits* of x.

With $d_i := b_{\overline{i}-i}$, we get $d_0 \neq 0$ and

$$x = \sum_{i=\underline{i}}^{\overline{i}} b_i 2^i = \sum_{i=0}^{\overline{i}-\underline{i}} b_{\overline{i}-i} 2^{\overline{i}-i} = \sum_{i=0}^{\overline{i}-\underline{i}} d_i 2^{-i} \cdot 2^{\overline{i}}.$$

This implies that $x \in \mathcal{F}^*(2, p, e_{\min}, e_{\max})$ if and only if $\overline{i} - \underline{i} < p$ and $e_{\min} \leq \overline{i} \leq e_{\max}$. 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

[18]by the formula $\sum_{i=0}^{n} x^i = (x^{n+1} - 1)/(x - 1)$ for $x \neq 1$

[19]Negative numbers are dealt with by the sign bit s.

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

$$x = \sum_{i=-\infty}^{0} b_i 2^i = b_0 + \sum_{i=-\infty}^{-1} b_i 2^i = b_0 + \sum_{i=-\infty}^{0} b_{i-1} 2^{i-1} = b_0 + \frac{1}{2} \underbrace{\sum_{i=-\infty}^{0} b_{i-1} 2^i}_{=:x'}.$$

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.

$$\begin{array}{rcccccl}
 & & & & 1.1 & \rightarrow & b_0 = 1 \\
2(1.1-1) & = & 2 \cdot 0.1 & = & 0.2 & \rightarrow & b_{-1} = 0 \\
2(0.2-0) & = & 2 \cdot 0.2 & = & 0.4 & \rightarrow & b_{-2} = 0 \\
2(0.4-0) & = & 2 \cdot 0.4 & = & 0.8 & \rightarrow & b_{-3} = 0 \\
2(0.8-0) & = & 2 \cdot 0.8 & = & 1.6 & \rightarrow & b_{-4} = 1 \\
2(1.6-1) & = & 2 \cdot 0.6 & = & 1.2 & \rightarrow & b_{-5} = 1 \\
2(1.2-1) & = & 2 \cdot 0.2 & = & 0.4 & \rightarrow & b_{-6} = 0 \\
 & & & & \vdots & &
\end{array}$$

We now see that the binary expansion of the decimal number 1.1 is periodic: the corresponding binary number is $1.0\overline{0011}$, and it has infinitely many significant digits. Since all numbers in the floating point number systems $\mathcal{F}(2, p, e_{\min}, e_{\max})$ and $\mathcal{F}^*(2, p, e_{\min}, e_{\max})$ 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, $e_{\min}$ and $e_{\max}$. 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 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 easily have occurred 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 $\mathcal{F}^*(2, p, e_{\min}, e_{\max})$, 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

$$x = \sum_{i=-\infty}^{\bar{\imath}} b_i 2^i = b_{\bar{\imath}}.b_{\bar{\imath}-1} \ldots \cdot 2^{\bar{\imath}}, \quad \text{where } b_{\bar{\imath}} = 1.$$

There are two natural ways of approximating x with p or less significant digits. One way is to round down, resulting in the number

$$\underline{x} = b_{\bar{\imath}}.b_{\bar{\imath}-1} \ldots b_{\bar{\imath}-p+1} \cdot 2^{\bar{\imath}} = \sum_{i=\bar{\imath}-p+1}^{\bar{\imath}} b_i 2^i.$$

This truncates all the digits b_i, $i \leq \bar{\imath} - p$, and the error we make is

$$x - \underline{x} = \sum_{i=-\infty}^{\bar{\imath}-p} b_i 2^i \leq \sum_{i=-\infty}^{\bar{\imath}-p} 2^i = 2^{\bar{\imath}-p+1}.$$

Alternatively, we could round up to the number[20]

$$\overline{x} = \underline{x} + 2^{\bar{\imath}-p+1}$$

[20] We have to check that this number has at most p significant digits. This is true if $b_{\bar{\imath}-p+1} = 0$, since then the addition of $2^{\bar{\imath}-p+1}$ adds exactly one digit to the at most $p-1$ significant digits of $\underline{x}$. And if $b_{\bar{\imath}-p+1} = 1$, the addition of $2^{\bar{\imath}-p+1}$ removes the least significant coefficient of $2^{\bar{\imath}-p+1}$ and *may* create one extra carry digit at the other end.

where our previous error estimate shows that indeed, $x \leq \overline{x}$ holds.

This means that x is between two numbers that are $2^{\bar{i}-p+1}$ apart, so the nearer of the two numbers is at most $2^{\bar{i}-p}$ away from x. On the other hand, x has size *at least* $2^{\bar{i}}$, meaning that

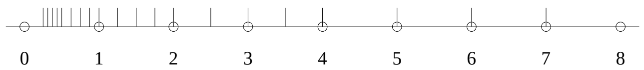
$$|x - \hat{x}|/x \leq 2^{-p},$$

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.



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 $\{e_{\min}, \ldots, e_{\max}\}$. 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

$$\begin{array}{rr} & 1.111 \cdot 2^{-2} \\ + & 1.011 \cdot 2^{-1} \ . \end{array}$$

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:

$$\begin{array}{rr} & 1.111 \cdot 2^{-2} \\ + & 10.110 \cdot 2^{-2} \ . \end{array}$$

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 form a memory cell. 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 $254 = 2^8 - 2$ possible values, and another bit is needed for the sign.

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.[21] 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 any 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 any 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.C` 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.

[21] this is called *round-to-even*; other rounding modes can be enabled if necessary.

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 β^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. 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.C` shown below computes the two sums and outputs them.

```
1  // Program: harmonic.C
2  // Compute the n-th harmonic number in two ways.
3
4  #include <iostream>
5
6  int main()
7  {
8    // Input
9    std::cout << "Compute H_n for n =? ";
10   unsigned int n;
11   std::cin >> n;
12
13   // Forward sum
14   float fs = 0;
15   for (unsigned int i = 1; i <= n; ++i)
16     fs += 1.0f / i;
17
18   // Backward sum
19   float bs = 0;
20   for (unsigned int i = n; i >= 1; --i)
21     bs += 1.0f / i;
22
23   // Output
24   std::cout << "Forward sum  = " << fs << "\n"
25             << "Backward sum = " << bs << "\n";
26   return 0;
27 }
```

Program 12: *progs/harmonic.C*

Think for a second and recall why it is important to *not* write `1 / i` in line 16 and line 21. Now let us have a look at an execution of the program.

```
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 $\frac{1}{i}$ to be added is so large that the addition does not change the partial sum anymore, just like in $2^p + 1$ "=" 2^p. 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

```
float b = 16777216.0f;  // 2^24
float a =  8388607.0f;  // 2^23 - 1
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.C` in Section 2.2.5. Here are the relevant expressions together with their meanings, shown for the type `float`.

expression (of type int)	meaning
`std::numeric_limits<float>::radix`	β
`std::numeric_limits<float>::digits`	p
`std::numeric_limits<float>::min_exponent`	$e_{min}+1$
`std::numeric_limits<float>::max_exponent`	$e_{max}+1$

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 `NaN`s (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;

(G3) compute the floating point number nearest to a given real number, with respect to a finite floating point number system;

(G4) work with a given floating point number system;

(G5) recognize usage of floating point numbers that violates any of the three Floating Point Arithmetic Guidelines;

(G6) write programs that perform computations with floating point numbers.

2.5.10 Exercises

Exercise 42 *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.* *(G1)*

a) `6 / 4 * 2.0f - 3`

b) `2 + 15.0e7f - 3 / 2.0 * 1.0e8`

c) `392593 * 2735.0f - 8192 * 131072 + 1.0`

d) `16 * (0.2f + 262144 - 262144.0)`

Exercise 43 *Compute the binary expansions of the following decimal numbers.*
a) 0.25 *b)* 1.52 *c)* 1.3 *d)* 11.1 *(G2)*

Exercise 44 *For the numbers in Exercise 43, compute nearest floating point numbers in the systems* $\mathcal{F}^*(2, 5, -1, 2)$ *and* $\mathcal{F}(2, 5, -1, 2)$. *(G3)*

Exercise 45 *What are the largest and smallest* positive *normalized single and double precision floating point numbers, according to the IEEE standard 754?* *(G4)*

Exercise 46 *How many floating point numbers do the systems* $\mathcal{F}^*(\beta, p, e_{min}, e_{max})$ *and* $\mathcal{F}(\beta, p, e_{min}, e_{max})$ *contain?* *(G4)*

Exercise 47 *Compute the value of the variable* `d` *after the declaration statement*

```
float d = 0.1;
```

Assume the IEEE standard 754. *(G3)*

Exercise 48 *What is the (potential) problem with the following loop?* (G5)

```
for (float i = 0.1; i != 1.0; i += 0.1)
  std::cout << i << "\n";
```

Exercise 49 *What is the (potential) problem with the following loop?* (G5)

```
for (float i = 0.0f; i < 100000000.0f; ++i)
   std::cout << i << "\n";
```

Exercise 50 *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* $\beta = 2$. *(G3)(G6)*

Exercise 51 *The number* π *can be defined through various infinite sums. Here are two of them.*

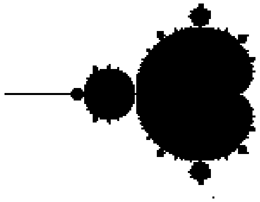
$$\begin{aligned}\frac{\pi}{4} &= 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots \\ \frac{\pi}{2} &= 1 + \frac{1}{3} + \frac{1 \cdot 2}{3 \cdot 5} + \frac{1 \cdot 2 \cdot 3}{3 \cdot 5 \cdot 7} + \cdots\end{aligned}$$

Write a program for computing an approximation of π, *based on these formulas. Which formula is better for that purpose?* *(G6)*

Exercise 52 *Write a program* `fpsys.C` *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 46.* *(G4)(G6)*

2.5.11 Challenges

Exercise 53 *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 54 *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.*