2.5 Floating point numbers

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 platforms, due to specialised 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 that 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

\[ \sum_{i=3}^{10} \beta_i 10^i, \]

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 realise this is to store an exponent such that the represented number is of the form

\[ \text{significand} \times 10^{\text{exponent}}. \]

For example,

\[ 82.4 = 824 \times 10^{-1}, \]
\[ 0.0824 = 824 \times 10^{-4}. \]

2.5.1 The types float and double

The types float and double are fundamental types provided by C++, and they store numbers in floating point representation.

While the fundamental types int and unsigned int are meant to approximate the "mathematical types" \( \mathbb{Z} \) and \( \mathbb{N} \), respectively, the goal of both float and double is to approximate the set \( \mathbb{R} \) of real numbers. Since there are much more real numbers than integers, this goal seems even more ambitious (and less realistic) than trying to approximate \( \mathbb{Z} \), say, with a finite value range. Nevertheless, the two types float and double are very useful in practical applications. The floating point representation allows values that are much larger than any value of type int and unsigned int. In fact, the value ranges of the floating point number types float and double are sufficient in most applications.

Values of these two types are referred to as floating point numbers, where double usually allows higher (namely, double) precision in approximating real numbers.

On the types float and double we have the same arithmetic, relational, and assignment operators as on integral types, with the same associativities and precedences. The only exception is that the modulus operators \( \% \) and \( \%\% \) are available for integral types only. This makes sense, since division over float and double is meant to model the true division over \( \mathbb{R} \) which has no remainder.

Like integral types, the floating point number types are arithmetic types, and this completes the list of fundamental arithmetic types in C++.

Literals of type float and double. Literals of types float and double are more complicated than literals of type int or unsigned int. For example, \( 1.23e-7 \) is a valid double literal, representing the value \( 1.23 \times 10^{-7} \). 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.
1 // Program: euler.C
2 // approximate Euler's constant e
3 #include <iostream>
4
5 int main ()
6 {
7    // values for term i, initialized for i = 0
8    float t = 1.0f; // 1/s!
9    float e = 1.0f; // i-th approximation of e
10   std::cout << "Approximating the Euler constant...\n";
11   // steps 1,...,n
12   for (unsigned int i = 1; i < 10; ++i) {
13       e += t /= i; // compact form of t = t / i; e = e + t
14       std::cout << "Value after term " << i + 1 << ": " << e << "\n";
15    }
16    return 0;
17 }

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
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 46), 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 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 Celsius leads to the behavior we want: during evaluation of the expression 9 + celsius / 5.0, 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 is

bool < int < unsigned int < float < double,

Value after term 2: 2.5
Value after term 3: 2.66667
Value after term 4: 2.70833
Value after term 5: 2.71667
Value after term 6: 2.71806
Value after term 7: 2.71825
Value after term 8: 2.71828
Value after term 9: 2.71828

It seems that we do get a good approximation of the Euler constant in this way. What remains to be explained is how the mixed expression e += t /= i in line 15 is dealt with that contains operands of types unsigned int and float. Note that since the arithmetic assignment operators are right-associative (Table 1 on Page 46), 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 FLOATING POINT NUMBERS

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 \texttt{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;  // Obtain the int value resulting from the float value –1.6, we can
```

We can simply write the expression \( int(-1.6f) \). The general syntax of an explicit conversion, also called a cast expression.

```
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 \( \text{T expr} \)) 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 result 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 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 \( \times 10^{308} \), a pretty large number.

Let us approach the issue of holes with a very simple program that asks the user to input two floating point numbers and their difference. The program then checks whether this is indeed the correct difference. Program 11 performs this task.

```c
// 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: \texttt{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.
2.5, FLOATING POINT NUMBERS

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 floating point number system is a finite subset of 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 $\mathbb{F}^*(\beta, p, e_{\min}, e_{\max})$ 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 $e \in \{e_{\min}, \ldots, e_{\max}\}$, and the number $e$ is the exponent.

We typically write a floating point number in the form

$$\pm d_0 d_1 \ldots d_{p-1} \beta^e.$$

For example, using base $\beta = 10$, the number 0.1 can be written as $1.0 \cdot 10^{-1}$, and as $0.1 \cdot 10^{1}$, and in many other ways.

The representation of a number becomes unique when we restrict ourselves to the set $\mathbb{F}^*(\beta, p, e_{\min}, e_{\max})$ 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 $\beta^{-m}$ (see also Exercise 30).

For a fixed exponent $e$, the smallest positive normalized number is

$$1.0 \ldots \cdot \beta^{-1},$$

while the largest one is

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

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.

An older equivalent term is mantissa.

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

Computing the floating point representation. In order to convert a given positive $2^b$ decimal number $x$ into a normalized binary floating point number system $\mathbb{F}^*(2, p, e_{\min}, e_{\max})$, 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

$$1.25 = 1.0 \cdot 2^{-2} + 1.0 \cdot 2^0.$$

We then determine the smallest and largest values of $i, j$ and $l$, for which $b_i$ is nonzero (note that $j$ may be $-\infty$, but $l$ is finite since $x$ is finite). The number $l+1 \in \mathbb{N} \cup \{\infty\}$ is the number of significant digits of $x$.

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

$$x = \sum_{i=-\infty}^{\infty} b_i 2^i = \sum_{i=0}^{\infty} b_i 2^{i-1} = \sum_{i=0}^{\infty} d_i 2^{-i} 2^i.$$

This implies that $x \in \mathbb{F}^*(2, p, e_{\min}, e_{\max})$ if and only if $l+1 < p$ and $e_{\min} \leq l \leq e_{\max}$. Equivalently, if the binary expansion of $x$ has at most $p$ significant digits, and the exponent of the normalized representation is within the allowable range.

In computing the binary expansion of $x > 0$, let us assume for simplicity that $x < 2$. This is sufficient to explain the issue with the decimal numbers 1.1 and 0.1, and all other cases can be reduced to this case by separately dealing with the largest even integer smaller or equal to $x$: writing $x = y + 2k$ with $k \in \mathbb{N}$ and $y < 2$, we get the binary expansion of $x$ by combining the expansions of $y$ and $2k$.

For $x < 2$, we have

$$x = \sum_{i=-\infty}^{0} b_i 2^i = b_0 + \sum_{i=-\infty}^{-1} b_i 2^i = b_0 + \sum_{i=0}^{\infty} b_i 2^{-i+1} = b_0 + 1 \sum_{i=-\infty}^{0} b_i 2^i.$$
where our previous error estimate shows that indeed, \( x \leq 2 \) holds.

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

\[
|x - \xi|/|x| \leq 2^{-p},
\]

where \( \xi \) is the floating point number nearest to \( x \). The number \( 2^{-p} \), referred to as the \textit{machine epsilon}, is the relative error made in approximating \( x \) with its nearest floating point number \( \xi \).

The previous inequality also holds for negative \( x \) and their corresponding best approximation \( \xi \), so that we get the general relative error formula

\[
|x - \xi|/|x| \leq 2^{-p}, \quad x \neq 0.
\]

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 \( \mathcal{F} \{2, 3, -2, 2\} \). The smallest positive number is 1.00-2^{-1} = 1/4, and the largest one is 1.11-2^{7} = 7 (recall that the digits are binary). The distribution of all possible numbers over the interval \([1/4, 7]\) is shown in the following picture.

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

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 \( \xi \) nearest to \( x \) is in fact a floating point number in the system we consider, i.e. if \( x \) has its exponent in the allowed range \([e_{\text{min}}, e_{\text{max}}]\). This fails if \( \xi \) 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).
2.5. FLOATING POINT NUMBERS

In practice, floating point number arithmetic is not more difficult than integer arithmetic, let us illustrate this with an example. Suppose that \( p = 4 \), and that we have a binary system; we want to perform the addition:

\[
1.111 \cdot 2^{-2} + 1.011 \cdot 2^{-1}.
\]

The first step is to align the two numbers such that they have the same exponent. This means to "normalize" one of the two numbers, e.g., the second one:

\[
1.111 \cdot 2^{-2} + 10.110 \cdot 2^{-2}.
\]

Now we can simply add up the two significands, just like we add integers in binary representation. The result is:

\[
100.101 \cdot 2^{-2}.
\]

Finally, we renormalize and obtain:

\[
1.00101 \cdot 2^{0}.
\]

We now realize that this exact result is not representable with \( p = 4 \) significant digits, so we have to round. In this case, the nearest representable number is obtained by simply dropping the last two digits:

\[
1.001 \cdot 2^{0}.
\]

2.5.6 The IEEE standard 754

Value range. The C++ standard does not prescribe the value range of the types `float` and `double`, it only stipulates that the value range of `float` is contained in the value range of `double` such that a `float` value can be promoted to a `double` value. In practice, most platforms support (variants of) the IEEE standard 754 for representing and computing with floating point numbers. Under this standard, the value range of the type `float` is the set

\[
F^*(2, 24, -126, 127)
\]

of single precision normalized floating point numbers, plus some special numbers (conveniently, 0 is one of these special numbers). The value range of `double` is the set

\[
F^*(2, 53, -1022, 1023)
\]

of double precision normalized floating point numbers, again with some special numbers added, including 0.

These parameters may seem somewhat arbitrary at first, but they are motivated by a common memory layout in which 32 bits 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 \( 2^{23} = 2^8 - 2 \) 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 \( 2^{52} = 2^{11} - 2 \) 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 \( d_{n-1} = 0 \) is chosen.\(^2\) 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.

\(^2\)This is called "round to even"; other rounding modes can be enabled if necessary.
2.5. FLOATING POINT NUMBERS

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 \( x \times y \times y \) and \( (x + y) \times (x - y) \). Therefore, testing the results of two floating point computations for equality using the relational operators \( = \) or \( \neq \) 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 significant, 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 \( 2^p \). 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 \( 2^p + 1 \) and 1. What will be the result? Mathematically, it is

\[
2^p + 1 = \sum_{i=0}^p b_i 2^i,
\]

with \((b_0, b_{p-1}, \ldots, b_0) = (1, 0, \ldots, 0, 1)\). 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 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.

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

Program 12: pros/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
2.5. FLOATING POINT NUMBERS

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 \( \frac{1}{2} \) to be added is so large that the addition does not change the partial sum anymore, just like in \( 2^n + 1 \sim 2^{n+1} \). 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 significant 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

\[
a x^2 + bx + c = 0, \quad a \neq 0.
\]

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 \( d \) 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 \( b = 2^a, a = 2^{p-1} - 1, c = 2^{p-1} + 1 \) (all these numbers are exactly representable). Then the exact value of \( d \) is \( -4 \). The value \( b^2 = 2^{2a} \) is a representable floating point number, but \( 4ac = 2^{2b} - 4 \) 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 1), and after the (error-free) subtraction, we get \( d = 0 \). 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

```c++
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.
2.5, Floating Point Numbers

You will most likely see the cancellation effect in the following seemingly equivalent variant of the above code,

```cpp
def b = 16777216.0f; // 2^24
def a = 8388607.0f; // 2^23 - 1
def c = 8388609.0f; // 2^23 + 1
```

```cpp
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 ac 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 author, for example, long double corresponds to the normalized floating point number system \(\mathbb{F}([2, 64, -16382, 16384])\). 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.h in Section 2.2.5. Here are the relevant expressions together with their meanings, shown for the type float,

<table>
<thead>
<tr>
<th>expression (of type int)</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>std::numeric_limits&lt;float&gt;::radix</td>
<td>b</td>
</tr>
<tr>
<td>std::numeric_limits&lt;float&gt;::digits</td>
<td>p</td>
</tr>
<tr>
<td>std::numeric_limits&lt;float&gt;::min_exponent</td>
<td>(c_{\text{min}} + 1)</td>
</tr>
<tr>
<td>std::numeric_limits&lt;float&gt;::max_exponent</td>
<td>(c_{\text{max}} + 1)</td>
</tr>
</tbody>
</table>

We remark that std::numeric_limits<float>::min() does not give the smallest float value (because of the sign bit, this smallest value is simply the negative of the largest value), but the smallest normalized positive value.

Special numbers. We have mentioned that the floating point systems prescribed by the IEEE standard 754 contain some special numbers; their encoding uses exponent values that do not occur in normalized numbers.

On the one hand, there are the denormalized numbers of the form

\[ \pm d_0 d_1 \ldots d_{p-1} \beta^{-m} \]

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 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.
2.5, FLOATING POINT NUMBERS

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 35 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.

\(\text{(G1)}\)

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 36 Compute the binary expansions of the following decimal numbers.

\(\text{(G2)}\)

a) \(0.25\)  b) \(1.52\)  c) \(1.3\)  d) \(11.1\)

Exercise 37 For the numbers in Exercise 36, compute nearest floating point numbers in the systems $F(2, 5, -1, 2)$ and $F(2, 5, -1, 2)$.

\(\text{(G3)}\)

Exercise 38 What are the largest and smallest positive normalized single and double precision floating point numbers, according to the IEEE standard 754?

\(\text{(G4)}\)

Exercise 39 How many floating point numbers do the systems $F(\beta, p, e_{min}, e_{max})$ and $F(\beta, p, e_{min}, e_{max})$ contain?

\(\text{(G4)}\)

Exercise 40 Compute the value of the variable \(d\) after the declaration statement

\(\text{float d = 0.1;}\)

Assume the IEEE standard 754.

\(\text{(G3)}\)

Exercise 41 What is the (potential) problem with the following loop?

\(\text{for (float i = 0.1; i != 1.0; i += 0.1)}\)

\(\text{std::cout << i << \"\n\";}\)

Exercise 42 What is the (potential) problem with the following loop?

\(\text{for (float i = 0.0f; i < 100000000.0f; ++i)}\)

\(\text{std::cout << i << \"\n\";}\)

Exercise 43 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 \(\beta = 2\).

\(\text{(G3)/(G6)}\)

Exercise 44 The number \(\pi\) can be defined through various infinite sums. Here are two of them,

\[\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots\]

\[\frac{\pi}{2} = 1 + \frac{1}{3} + \frac{1.2}{3.5} + \frac{1.2.3}{3.5.7} + \ldots\]

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

\(\text{(G6)}\)

Exercise 45 Write a program tpeacs.C to visualize a normalized floating point number system $F(2, p, e_{min}, e_{max})$. The program should read the parameters \(p\), \(e_{min}\) and \(e_{max}\) as inputs and for each positive number \(x\) from $F(2, p, e_{min}, e_{max})$ 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 39.

\(\text{(G4)/(G6)}\)