2.2 Integers

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

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

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

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

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

```
1  // Program: fahrenheit.C
2  // Convert temperatures from Celsius to Fahrenheit.
3
4  #include <iostream>
5
6  int main()
7  {
8    // Input
9    std::cout << "Temperature in degrees Celsius =? ";
10   int celsius;
11   std::cin >> celsius;
12
13   // Computation and output
14   std::cout << celsius << " degrees Celsius are "
15             << 9 * celsius / 5 + 32 << " degrees Fahrenheit.\n";
16   return 0;
17 }
```

Program 4: *progs/fahrenheit.C*

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

```
15 degrees Celsius are 59 degrees Fahrenheit.
```


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

```
9 * celsius / 5 + 32
```

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

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

2.2.1 Associativity and precedence of operators

The evaluation of an expression is to a large extent governed by the *associativities* and *precedences* of the involved operators. In short, associativities and precedences determine the logical parentheses in an expression that is not, or only incompletely, parenthesized. We have already touched associativity in connection with the output operator in Section 2.1.13.

C++ allows incompletely parenthesized expressions in order to save parentheses at obvious places. This is like in mathematics, where we write $3 + 4 \cdot 5$ when we mean $3+(4\cdot5)$. We also write $3+4+5$, even though it is not a priori clear whether this means $(3+4)+5$ or $3+(4+5)$. Here, the justification is that addition is *associative*, so it does not matter which variant we mean.

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

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

The expression `9 * celsius / 5 + 32` involves the multiplication operator `*`, the division operator `/`, and the addition operator `+`. All three are binary operators. In C++ as in mathematics, the multiplicative operators `*` and `/` have *higher* precedence than the additive operators `+` and `-`. We also say that multiplicative operators *bind* more strongly than additive ones.[4] This means, our expression contains the logical paren-

[4] In American English, this rule is known as "PEMDAS", in British English it's "BODMAS", and in German it's "Punkt- vor Strichrechnung".

theses `(9 * celsius / 5) + 32`: it is a composite expression built from the addition operator and its operands `9 * celsius / 5` and `32`.

> **Arithmetic Evaluation Rule 2:** Binary arithmetic operators are left associative.

In mathematics, it does not matter how the sub-expression `9 * celsius / 5` is parenthesized. But in C++, it is done from *left to right*, that is, the two leftmost sub-expressions are grouped together. This is a consequence of the fact that the binary arithmetic operators are defined to be *left* associative. The expression `9 * celsius / 5` is therefore logically parenthesized as `(9 * celsius) / 5`, and our original expression has to be read as

```
((9 * celsius) / 5) + 32
```

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

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

The correct logical parentheses are

$$((-3) - 4),$$

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

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

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

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

2.2.2 Expression trees

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

The recursive structure of an expression can nicely be visualized in the form of an *expression tree*. In Figure 3, the expression tree for the expression `9 * celsius / 5 + 32` is shown.

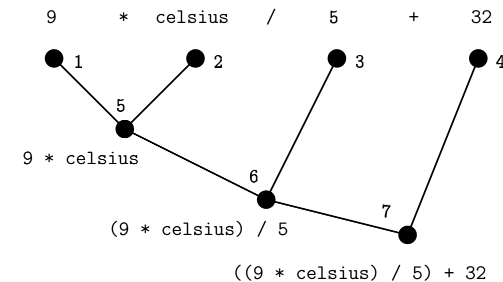

Figure 3: *An expression tree for* `9 * celsius / 5 + 32` *and its logical parenthization* `((9 * celsius) / 5) + 32`. *Nodes are labeled from one to seven.*

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

2.2.3 Evaluating expressions

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

When we associate the evaluation sequence with the corresponding sequence of nodes in the tree, a valid node sequence *topologically sorts* the tree. This means that any node in the sequence occurs only after all its children have occurred. In Figure 3, for example, the node sequence $(1, 2, 5, 3, 6, 4, 7)$ induces a valid evaluation sequence. Assuming that

the variable `celsius` has value 15, we obtain the following evaluation sequence. (In each step, the sub-expression to be evaluated next is marked by a surrounding box.)

$$\begin{aligned}
\boxed{9} * \texttt{celsius} / 5 + 32 &\longrightarrow^1 9 * \boxed{\texttt{celsius}} / 5 + 32 \\
&\longrightarrow^2 \boxed{9 * 15} / 5 + 32 \\
&\longrightarrow^5 135 / \boxed{5} + 32 \\
&\longrightarrow^3 \boxed{135 / 5} + 32 \\
&\longrightarrow^6 27 + \boxed{32} \\
&\longrightarrow^4 \boxed{27 + 32} \\
&\longrightarrow^7 59
\end{aligned}$$

The sequence $(1, 2, 3, 4, 5, 6, 7)$ is another valid node sequence, inducing a different evaluation sequence; the resulting value of 59 is the same. There are much more evaluation sequences, of course, and it is unspecified by the C++ standard which one is to be used.

In our small example, all possible evaluation sequences will result in value 59, but it is also not hard to write down expressions whose values and effects depend on the evaluation sequence being chosen (see Exercise 2*(g)*, Exercise 10*(h)*, and the Details section below). A program that contains such an expression might exhibit unspecified behavior. But through good programming style, this issue is easy to avoid, since it typically only occurs when one tries to squeeze too much functionality into a single line of code.

2.2.4 Arithmetic operators on the type int

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

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

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

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

```
9 * celsius / 5 + 32
```

by the expression

```
9 / 5 * celsius + 32
```


Description	Operator	Arity	Prec.	Assoc.
post-increment	++	1	17	left
post-decrement	--	1	17	left
pre-increment	++	1	16	right
pre-decrement	--	1	16	right
sign	+	1	16	right
sign	-	1	16	right
multiplication	*	2	14	left
division (integer)	/	2	14	left
modulus	%	2	14	left
addition	+	2	13	left
subtraction	-	2	13	left
assignment	=	2	4	right
mult assignment	*=	2	4	right
div assignment	/=	2	4	right
mod assignment	%=	2	4	right
add assignment	+=	2	4	right
sub assignment	-=	2	4	right

Table 1: *Arithmetic and assignment operators for the type* `int`*. Each increment or decrement operator expects an lvalue. The composite expression is an lvalue (pre-increment and pre-decrement), or an rvalue (post-increment and post-decrement). Each assignment operator expects an lvalue as left operand and an rvalue as right operand; the composite expression is an lvalue. All other operators involve rvalues only and have no effects.*

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

```
15 degrees Celsius are 47 degrees Fahrenheit.
```

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

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

$$a = (a \operatorname{div} b)b + a \bmod b$$

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

```
(a / b) * b + a % b
```

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

If both `a` and `b` have non-negative values, then `a % b` has a non-negative value as well. This implies that the integer division rounds *down* in this case. If (at least) one of `a` or `b` has a negative value, it is implementation defined whether division rounds up or down.[5] Note that by the identity `(a / b) * b + a % b`, the rounding mode for division also determines the functionality of the modulus operator. If `b` has value 0, the values of `a / b` and `a % b` are undefined.

Coming back to our example (and taking precedences and associativities into account), we get the following valid evaluation sequence for our alternative Celsius-to-Fahrenheit conversion.[6]

```
9 / 5 * celsius + 32 ⟶ 1 * celsius + 32
                     ⟶ 1 * 15 + 32
                     ⟶ 15 + 32
                     ⟶ 47
```

Here we see the "error" made by the integer division: `9 / 5` has value 1.

[5] There is a remark in the standard that future revisions may prescribe a rounding towards zero for these cases.

[6] To avoid longish evaluation sequences, we will from now on suppress the evaluation of literals.

Unary additive operators. We have already touched the unary - operator, and this operator does what one expects: the value of the composite expression -*expr* is the negative of the value of *expr*. There is a unary + operator, for completeness, although its "functionality" is non-existing: the value of the composite expression +*expr* is the same as the value of *expr*.

Increment and decrement operators. Each of the tokens ++ and -- is associated with two *distinct* unary operators that differ in precedence and associativity.

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

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

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

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

You may argue that the increment and decrement operators are superfluous, since their functionality can be realized by combining the assignment operator (Section 2.1.13) with an additive operator. Indeed, if `a` is a variable, the expression ++`a` is equivalent in value and effect to the expression `a = a + 1`. There is one subtlety, though: if *expr* is a general lvalue, ++*expr* is *not* necessarily equivalent to *expr* = *expr* + 1. The reason is that in the former expression, *expr* is evaluated only once, while in the latter, it is evaluated *twice*. If *expr* has an effect, this can make a difference.

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

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

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

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

Assignment operators. The assignment operator `=` is available for all types, see Section 2.1.13. But there are specific operators that combine the arithmetic operators with an assignment. These are the binary operators `+=`, `-=`, `*=`, `/=` and `%=`. The expression *expr1* `+=` *expr2* has the effect of adding the value of *expr2* (an rvalue) to the value of *expr1* (an lvalue). The object referred to by *expr1* is returned. This is a generalization of the pre-increment: the expression `++`*expr* is equivalent to *expr* `+= 1`. As before, *expr1* `+=` *expr2* is *not* equivalent to *expr1* `=` *expr1* `+` *expr2* in general, since the latter expression evaluates *expr1* twice.

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

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

2.2.5 Value range

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

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

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

of 2^b numbers.[7] You can find out the smallest and largest `int` values on your computer, using the library `limits`. The corresponding code is given in Program 5.

[7]The C++ standard does not prescribe this, but any different choice of value range would be somewhat unreasonable, given other requirements imposed by the standard.


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

Program 5: *progs/limits.C*

When you run the program `limits.C` on a 32-bit system, you will most likely get the following output.

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

Indeed, as $2147483647 = 2^{31}-1$, you can deduce that the number of bits used to represent an `int` value on this system is 32. At this point, you are not supposed to understand the expression `std::numeric_limits<int>::min()` in detail, but we believe that you get its idea.

It is clear that the arithmetic operators (except the unary `+` and the binary `/` and `%`) cannot work exactly like their mathematical counterparts, even when their arguments are restricted to representable `int` values. The reason is that the values of composite expressions constructed from these operators can under- or overflow the value range of the type `int`. The most obvious such example is the expression `2147483647+1`. As we have just seen, its mathematically correct value of 2147483648 is not representable over the type `int` on your system, so you will inevitably get some other value.

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

2.2.6 The type unsigned int

An object of type `int` can have negative values, but often we only work with natural numbers.[8] Using a type that represents only non-negative values allows to extend the range of positive values without using more bits. C++ provides such a type, it is called `unsigned int`. On this type, we have all the arithmetic operators we also have for `int`,

[8]For us, the set $\mathbb{N}$ of natural numbers starts with 0, $\mathbb{N} = \{0, 1, 2, \ldots\}$.

with the same arities, precedences and associativities. Given a b-bit representation, the value range of `unsigned int` is the set

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

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

```
Minimum value of an unsigned int object is 0.
Maximum value of an unsigned int object is 4294967295.
```

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

2.2.7 Mixed expressions and conversions

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

```
17+17u ⟶ 17u+17u ⟶ 34u
```

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

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

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

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

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

2.2.8 Binary representation

Assuming b-bit representation, we already know that the type `int` covers the values

$$-2^{b-1}, \ldots, 2^{b-1} - 1,$$

while `unsigned int` covers

$$0, \ldots, 2^b - 1.$$

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

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

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

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

$$13 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3.$$

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

Conversion decimal → binary. The identity

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

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

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

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

$$\sum_{i=0}^{k} b_i 2^i = b_0 + 2 \sum_{i=0}^{k-1} b_{i+1} 2^i = \ldots = b_0 + 2(b_1 + 2(b_2 + 2(\cdots + 2b_k) \ldots))$$

For example, to convert the binary number $b_4b_3b_2b_1b_0 = 10100$ into decimal representation, we compute

$$(((b_4 \cdot 2 + b_3) \cdot 2 + b_2) \cdot 2 + b_1) \cdot 2 + b_0 = (((1 \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 0) \cdot 2 + 0 = 20.$$

Representing unsigned int values. Since any `unsigned int` value

$$n \in \{0, \ldots, 2^b - 1\}$$

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

n	representation
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Representing int values. A common way of representing `int` values using the same b bits goes as follows. If the value n is non-negative, we store the binary representation of n itself—a number from

$$\{0, \ldots, 2^{b-1} - 1\}.$$

That way we use all the b-bit patterns that start with 0.

If the value n is negative, we store the binary representation of $n + 2^b$, a number from

$$\{2^{b-1}, \ldots, 2^b - 1\}.$$

This yields the missing b-bit patterns, the ones that start with 1. For $b = 3$, the resulting representations are

n	representation
−4	100
−3	101
−2	110
−1	111
0	000
1	001
2	010
3	011

This is called the *two's complement* representation. In this representation, adding two `int` values n and n' is very easy: simply add the representations according to the usual rules of binary number addition, and ignore the overflow bit (if any). For example, to add -2 and -1 in case of $b = 3$, we compute

$$\begin{array}{r} 110 \\ +\quad 111 \\ \hline 1101 \end{array}$$

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

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

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

2.2.9 Integral types

There is a number of other fundamental types to represent signed and unsigned integers, see the Details section. These types may differ from `int` and `unsigned int` with respect to their value range. All these types are called *integral types*, and for each of them, all the operators in Table 1 (Page 42) are available, with the same arities, precedences, associativities and functionalities (up to the obvious limits dictated by the respective value ranges).

2.2.10 Details

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

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

Logically parenthesizing a general expression. Given an expression that consists of a sequence of operators and operands, we want to deduce the logical parentheses. For each operator in the sequence, we know its arity, its precedence (a number between 1 and 18, see Table 1 on Page 42 for the arithmetic operators), and its associativity (left or right). In case of a unary operator, the associativity specifies on which side of the operator its operand is to be found.

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

expression	x_1	op_1	x_2	op_2	x_3	op_3	op_4	x_4
arity		2		2		2	1	
precedence		4		13		13	16	
associativity		r		l		l	r	

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

In our example, the leading operand of op_3 is the subsequence $x_2 \ op_2 \ x_3$ to the left of op_3, since the next operator of lower precedence to the left of op_3 is op_1.

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

According to this definition, the secondary operand of op_3 is $op_4 \ x_4$ in our example.

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

Here is the table for our example again, enhanced with the subsequences of all four operators that are put in parentheses according to the rules just described.

expression	x_1	op_1	x_2	op_2	x_3	op_3	op_4	x_4
arity		2		2		2	1	
precedence		4		13		13	16	
associativity		r		l		l	r	
op_1	(x_1	op_1	x_2	op_2	x_3	op_3	op_4	x_4)
op_2			(x_2	op_2	x_3)			
op_3			(x_2	op_2	x_3	op_3	op_4	x_4)
op_4							(op_4	x_4)

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

$$(\ x_1 \ op_1 \ ((\ x_2 \ op_2 \ x_3 \ ) \ op_3 \ (op_4 \ x_4 \ ))).$$

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

expression	x_1	op_1	x_2	op_2	x_3
arity		2		2	
precedence		13		13	
associativity		r		l	
op_1	(x_1	op_1	x_2	op_2	x_3)
op_2	(x_1	op_1	x_2	op_2	x_3)

The resulting parenthesized expression is

$$((\ x_1 \ op_1 \ x_2 \ op_2 \ x_3 \ )),$$

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

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

No such rule exists for the signed integral types, meaning that over- and underflow are dealt with at the discretion of the compiler.

Sequences of + and -. We have argued above that it is usually clear which operators occur in an expression, even though some of them share their token. But since the characters + and - are heavily overused in operator tokens, special rules are needed to resolve the meanings of sequences of +'s, or of -'s.

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

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

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

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

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

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

```
++i + i
```

where we suppose that `i` is a variable of type `int`. If `i` is initially 5, say, then the value of the composite expression may in practice be 11 or 12. The result depends on whether

or not the effect of the left operand `++i` of the addition is processed before the right operand `i` is evaluated. The value of the expression `++i + i` is therefore unspecified by the C++ standard.

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

```
i = ++i + 1
```

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

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

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

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

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

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

Therefore, we only have an issue with expressions that have more than one effect. Hence, if you prefer not to worry about effect order, ensure that each expression that you write generates at most one effect. Expressions with more than one effect can make

sense, though, and they are ok, as long as some sequence points separate the effects and put them into a well-defined order. This is summarized in the following rule.

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

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

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

```
nextvalue = 5 * nextvalue + 3
```

could more compactly be written as

```
(nextvalue *= 5) += 3
```

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

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

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

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

2.2.11 Goals

Dispositional. At this point, you should ...

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

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

(G1) parenthesize and evaluate a given arithmetic expression involving operands of types `unsigned int` and `int`, the binary arithmetic operators +,-, *, /, %, and the unary -;

(G2) convert a given decimal number into binary representation and vice versa;

(G3) derive the *two's complement* representation of a given number in b-bit representation, for some $b \in \mathbb{N}$;

(G4) write programs whose output is determined by a fixed number of arithmetic expressions involving literals and input variables of types `int` and `unsigned int`;

(G5) determine the value range of integral types on a given machine (using a program).

2.2.12 Exercises

Exercise 8 *Parenthesize the following expressions and then evaluate them step by step. This means that types and values of all intermediate results that are computed during the evaluation should be provided.* (G1)

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

Exercise 9 *Which of the following character sequences are not legal expressions, and why? For the ones that are, give the logical parenthesization. (In order to avoid (misleading?) hints, we have removed the spaces that we usually include for the sake of better readability.)* (G1)

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

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

Exercise 10 *For all legal expressions from Exercise 9, provide a step-by-step evaluation, supposing that initially* a *has value* 5, b *has value* 2*, and the value of* c *is undefined. Which of the expressions result in unspecified or undefined behavior?* (G1)

Exercise 11 *Prove that for all* $a \geq 0$ *and* $b, c > 0$*, the following equation holds:*

$$(a \operatorname{div} b) \operatorname{div} c = a \operatorname{div} (bc).$$

Exercise 12 *Compute by hand binary representations of the following decimal numbers.* (G2)

a) 15 *b)* 172 *c)* 329 *d)* 1022

Exercise 13 *Compute by hand decimal representations of the following binary numbers.* (G2)

a) 110111 *b)* 1000001 *c)* 11101001 *d)* 101010101

Exercise 14 *Assuming a* 4*-bit representation, compute the binary two's complement representations of the following decimal numbers.* (G3)
a) 6 *b)* -4 *c)* -8 *d)* 9 *e)* -3

Exercise 15 *Write a program* `celsius.C` *that converts temperatures from degrees Fahrenheit into degrees Celsius.*

The initial output that prompts the user to enter the temperature in degrees Fahrenheit should also contain lower *and* upper *bounds for the allowed inputs. These bounds should be chosen such that no over- and underflows can occur in the subsequent computations, given that the user respects the bounds. You may for this exercise assume that the integer division rounds towards zero for all operands: for example,* `-5 / 2` *then rounds the exact result* -2.5 *to* -2.

The program should output the correct *result in degrees Celsius as a mixed rational number of the form* $x\frac{y}{9}$ *(meaning* $x + y/9$*), where* $x, y \in \mathbb{Z}$ *and* $|y| \leq 8$*. For example,* $13\frac{4}{9}$ *could be output simply as* `13 4/9`*. We also allow for example the output* `-1 -1/9` *(meaning* $-1 - 1/9 = -10/9$*).* (G4)

Exercise 16 *Write a program* `threebin.C` *that reads a (decimal) number* $a \geq 0$ *from standard input and outputs the* last three *bits of* a*'s binary representation. Fill up with leading zeros in case the binary representation has less than three bits.* (G4)

2.2.13 Challenges

Exercise 17 *Josephus was a Jewish military leader in the Jewish-Roman war of 66-73. After the Romans had invaded his garrison town, the few soldiers (among them Josephus) that had survived the killings by the Romans decided to commit suicide. But somehow, Josephus and one of his comrades managed to surrender to the Roman forces without being killed (Josephus later became a Roman citizen and well-known historian).*

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

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

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

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