2.2 Integers

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:

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

```cpp
// Program: fahrenheit.C
// Convert temperatures from Celsius to Fahrenheit.

#include <iostream>
#include <iomanip>

int main()
{
    // Input
    std::cout << "Temperature in degrees Celsius = " << std::endl;
    int celsius;
    std::cin >> celsius;

    // Computation and output
    std::cout << celsius << " degrees Celsius are ";
    return 0;
}
```

15 degrees Celsius are 59 degrees Fahrenheit.

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

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

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

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

2.2.1 Associativity and precedence of operators

The evaluation of an expression is to a large extent governed by the associativities and precedences of the involved operators. In short, associativities and precedences determine the logical parentheses in an expression that is not, or only incompletely, parenthesized. We have already touched associativity in connection with the output operator in Section 2.1.12.

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

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

**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.

This means, our expression contains the logical parentheses (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

---

8In American English, this rule is known as “PEMDAS”, in British English it is “BODMAS”, and in German it is “Punkte vor Strichrechnung.”
are grouped together. This is a consequence of the fact that the binary arithmetic operators are defined to be left associative. The expression \(9 * \text{celsius} / 5\) is therefore logically parenthesized as \((9 * \text{celsius}) / 5\), and our original expression has to be read as
\[ ((9 * \text{celsius}) / 5) * 32 \]

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 54 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.

<table>
<thead>
<tr>
<th>expression</th>
<th>(x_1)</th>
<th>(\text{op}_1)</th>
<th>(x_2)</th>
<th>(\text{op}_2)</th>
<th>(x_3)</th>
<th>(\text{op}_3)</th>
<th>(x_4)</th>
<th>(\text{op}_4)</th>
<th>(x_5)</th>
<th>(\text{op}_5)</th>
<th>(x_6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>arity</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>precedence</td>
<td>4</td>
<td>13</td>
<td>13</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>associativity</td>
<td>r</td>
<td>1</td>
<td>1</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 operator, and as the right hand side operand otherwise. The leading operand for \(\text{op}\) includes everything to the relevant side between \(\text{op}\), and the next operator of lower precedence than \(\text{op}\). In other words, everything in between these two operators is "grabbed" by the "stronger" operator.

In our example, the leading operand of \(\text{op}\), is the subexpression \(x_2 \text{ op}_2 \ x_3\) to the left of \(\text{op}\), since the next operator of lower precedence to the left of \(\text{op}\) is \(\text{op}\).

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 \(\text{op}\) includes everything to the relevant side between \(\text{op}\), and the next operator of the same or lower precedence than \(\text{op}\). 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 \(\text{op}\), is \(\text{op}_4 \ x_4\) in our example.

Finally, we put a pair of parentheses around the subexpression 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.

<table>
<thead>
<tr>
<th>expression</th>
<th>(x_1)</th>
<th>(\text{op}_1)</th>
<th>(x_2)</th>
<th>(\text{op}_2)</th>
<th>(x_3)</th>
<th>(\text{op}_3)</th>
<th>(x_4)</th>
<th>(\text{op}_4)</th>
<th>(x_5)</th>
<th>(\text{op}_5)</th>
<th>(x_6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>arity</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>precedence</td>
<td>4</td>
<td>13</td>
<td>13</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>associativity</td>
<td>r</td>
<td>1</td>
<td>1</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Now we simply put together all parentheses that we have obtained, taking their multiplicities into account. In our example we get the expression
\[
( x_1 \ \text{op}_1 ( ( x_2 \ \text{op}_2 \ x_3 ) \ \text{op}_3 ( \text{op}_4 \ x_4 ) ) )
\]

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.

\[
x_1 \ \text{op}_1 ( ( x_2 \ \text{op}_2 \ x_3 ) \ \text{op}_3 ( \text{op}_4 \ x_5 ) )
\]

The resulting parenthesized expression is
\[
(( x_1 \ \text{op}_1 \ x_2 \ \text{op}_2 \ x_3 ))
\]

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.

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, 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.

According to Table 1, the unary \(-\) has precedence 16 and is right associative. The binary \(-\) has precedence 13 and is left associative. Thus, the unary \(-\) binds more strongly, and we expect to get the logical parentheses \((-3) - 4\). Indeed, applying the rules above yields

\[
\text{expr.} + - 3 - 4 \\
\text{arity} & 1 & 2 \\
\text{proc.} & 16 & 13 \\
\text{assoc.} & r & 1 \\
\text{unary} - ( - 3 ) \\
\text{binary} - ( - 3 - 4 )
\]

This induces the logical parentheses

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

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

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

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

### 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 \(9 \times \text{celsius} / 5 + 32\) is shown.

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.

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

#### 2.2.3 Evaluating expressions

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

When we associate the evaluation sequence with the corresponding sequence of nodes in the tree, a valid node sequence topologically sorts the tree. This means that any node in the sequence occurs only after all its children have occurred. In Figure 3, for example, the node sequence \((1, 2, 3, 6, 4, 7)\) induces a valid evaluation sequence. Assuming that the variable \(\text{celsius}\) has value 15, we obtain the following evaluation sequence. (In each step, the sub-expression to be evaluated next is marked by a surrounding box.)

\[
\begin{array}{ll}
9 \times \text{celsius} / 5 + 32 & \rightarrow^1 9 \times [\text{celsius}] / 5 + 32 \\
& \rightarrow^2 9 \times 15 / 5 + 32 \\
& \rightarrow^3 135 / 5 + 32 \\
& \rightarrow^4 135 / 5 + 32 \\
& \rightarrow^5 27 + 32 \\
& \rightarrow^6 27 + 32 \\
& \rightarrow^7 59
\end{array}
\]

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 evalua-
2.2. INTEGERS

The division operator. According to the rules of mathematics, we could replace the expression
\[ 9 \times \text{celsius} / 5 + 32 \]
by the expression
\[ 9 / 5 \times \text{celsius} + 32 \]
without affecting its value and the functionality of the program 
\text{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 67 degrees Fahrenheit.

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

The modulus operator. The remainder of the integer division can be obtained with the binary modulus operator %, in mathematics denoted by mod. The mathematical rule
\[ a = (a \div b) \times b + a \mod b \]
also holds in C++: for example, if \(a\) and \(b\) are variables of type int, the value of \(b\) being non-zero, the expression
\[ (a / b) \times b + a \mod b \]
has the same value as \(a\). The modulus operator is considered as a multiplicative operator and has the same precedence (14) and associativity (left) as the other two multiplicative operators \(*\) and /.

If both \(a\) and \(b\) have non-negative values, then \(a \mod b\) has a non-negative value as well. This implies that the integer division rounds down in this case. If (at least) one of \(a\) or \(b\) has a negative value, it is implementation defined whether division rounds up or down.\(^6\) Note that by the identity \((a / b) \times b + a \mod b\), the rounding mode for division also determines the functionality of the modulus operator. If \(b\) has value 0, the values of \(a / b\) and \(a \mod b\) are undefined.

Coming back to our example (and taking precedences and associativities into account), we get the following valid evaluation sequence for our alternative Celsius-to-Fahrenheit conversion:\(^7\)

\(^6\) There is a remark in the standard that future revisions may prescribe a rounding towards zero for these cases.

\(^7\) To avoid longish evaluation sequences, we will from now on suppress the evaluation of literals.
2.2. **INT**EGERs

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.

**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 post-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 *post*-fix notation is illustrated in the following code fragment,

```cpp
int a = 7;
std::cout << ++a << "\n"; // outputs 8
std::cout << a++ << "\n"; // outputs 8
std::cout << a << "\n"; // outputs 9
```

You may argue that the increment and decrement operators are superfluous, since their functionality can be realised by combining the assignment operator (Section 2.1.12) 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 value, ++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 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 2.1.12. But there are specific operators that combine the arithmetic operators with an assignment. These are the binary operators +=, -=, *=, /= and %=.

The expression expr += 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 generalisation of the assignment: 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 expr2 twice.

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

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

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

of \(2^b\) numbers. 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.

\(\text{The C++ standard does not prescribe this, but any different choice of value range would be somewhat unreasonable, given other requirements imposed by the standard.}\)
2.2. \textbf{INT}EGERs

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 " << std::numeric_limits<int>::min() << 
10      "\n"
11   << std::numeric_limits<int>::max() << "\n";
12   return 0;
13 }

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 \textbf{The type unsigned int}

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

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

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 \texttt{U} or \texttt{u}. For example, \texttt{127u} and \texttt{0u} are valid literals of type unsigned int, with their values immediately apparent.

2.2.7 \textbf{Mixed expressions and conversions}

Expressions may involve sub-expressions of type int and of type unsigned int. For example \texttt{17u + 17u} is a valid 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 \texttt{17u + 17u} is of type unsigned int and gets evaluated step by step as

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

This might be somewhat confusing, since in mathematics, it is just the other way around; \( Z \) (the set of integers) is more general than \( N \) (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

\[
\begin{align*}
\text{int } & a = 3u; \\
& b = 4294967295u;
\end{align*}
\]

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

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

Conversion decimal \(\rightarrow\) binary. The identity
\[n = \sum_{i=0}^{\infty} b_i 2^i = b_0 + \sum_{i=0}^{\infty} b_{i+1} 2^{i+1} = b_0 + 2 \sum_{i=0}^{\infty} b_{i+1} 2^{i},\]
provides a simple algorithm to compute the binary representation of a given decimal number \(n \in \mathbb{N}\). The least significant coefficient \(b_0\) of the binary expansion of \(n\) is \(n \mod 2\). 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 \(b_0 = 14 \mod 2 = 0\) and \(n' = (14 - 0)/2 = 7\). We continue with \(n = 7\) and get \(b_1 = 7 \mod 2 = 1\) and \(n' = (7 - 1)/2 = 3\). For \(n = 3\) we get \(b_2 = 3 \mod 2 = 1\) and \(n' = (3 - 1)/2 = 1\) which leaves us with \(n = 1\). In summary, the binary representation of 14 is \(b_3b_2b_1b_0 = 1100\).

Conversion binary \(\rightarrow\) decimal. To convert a given binary number \(b_k \ldots b_0\) into decimal representation, we can once again use the identity from above,
\[
\sum_{i=0}^{k} b_i 2^i = b_0 + 2 \sum_{i=0}^{k-1} b_{i+1} 2^{i} = \ldots = b_0 + 2(b_1 + 2(b_2 + 2(\cdots + 2b_k)\ldots))
\]

For example, to convert the binary number \(b_3b_2b_1b_0 = 10100\) into decimal representation, we compute
\[(((b_3 \cdot 2 + b_2) \cdot 2 + b_1) \cdot 2 + b_0) = (((1 \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 0) = 20.\]

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:

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

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

If the value \(n\) is negative, we store the binary representation of \(n + 2^b\), a number from
\[\{2^{b-1}, \ldots, 2^b - 1\}.\]

This yields the missing \(b\)-bit patterns, the ones that start with 1. For \(b = 3\), the resulting representations are
### 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 $31 = 1 \cdot 16^1 + 15 \cdot 16^0$.

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 $2^n$, 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 $2^n$. For example, if a is a variable of type unsigned int with non-zero value, then $-a$ has value $2^n - a$.

No such rule exists for the signed integral types, meaning that overflow 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 overloaded 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 parenthesisation `--(-a)` is invalid for another reason: the operand of the pre-increment must be an value, but the expression `-a` is an rvalue.

### 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 45) are available, with the same arities, precedences, associativities and functionalities (up to the obvious limits dictated by the respective value ranges).

<table>
<thead>
<tr>
<th>n</th>
<th>representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td>100</td>
</tr>
<tr>
<td>-3</td>
<td>101</td>
</tr>
<tr>
<td>-2</td>
<td>110</td>
</tr>
<tr>
<td>-1</td>
<td>111</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>011</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
\text{110} & \quad \text{+} \quad \text{111} \\
\text{101} & \quad \text{=} \quad \text{111}
\end{align*}
\]

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.
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 integer 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.

For once, recall that the evaluation order for the sub-expressions of a given expression is not specified in general. For an expression to be valid it has to give the same value for all possible evaluation sequences. This is, for instance, not the case for the expression

\[ \text{++i } \text{ i} \]

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

Second, also the order in which the effects of an expression are processed is not specified. Obviously, this is only an issue if an expression has 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. In any case, all expressions have to obey the following rule:

**Single Modification Rule:** The evaluation of an expression may modify the value of an object of fundamental type at most once.

For example, consider the following innocent looking expression that involves a variable \( i \) of type int,

\[ i = \text{++i } i \]

This expression has two effects: the increment of \( i \) and the assignment to \( i \). As both effects modify the value of \( i \), the expression does not respect the Single Modification Rule and, therefore, it is invalid.

At first sight, this seems strange. The sub-expression \( \text{++i } i \) is well-defined, and it has to be evaluated before the assignment takes place. So where is the problem? It is a consequence of our underlying computer model, the von Neumann architecture. From the computer’s point of view, the evaluation of the sub-expression \( \text{++i } i \) 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 \( \text{++i } i \) and, hence, have to be processed before the assignment. But the third step does not necessarily have to be completed before the assignment. In order to allow the compiler to optimize the transfer of data between CPU registers and main memory (which is very much platform dependent), this order has not been specified. In fact, it is not unreasonable to assume that the traffic between registers and main memory is organized such that several items are transferred at once or quickly after another, using so-called bursts.

Suppose as before that \( i \) initially has value 5. If the assignment is performed after the register content is written back to main memory, \( i = \text{++i } i \) 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.

Fortunately, you do not have to worry about these details due to the Single Modification Rule. Also, it is guaranteed that all effects of an expression statement are processed before the next statement is executed. The semicolon that terminates an expression statement is said to form a sequence point. At any sequence point, all effects generated by preceding statements or expressions have been completed.

If you perceive the previous example as artificial, here is a "more natural" violation of the single modification rule: if \( \text{nextvalue} \) is a variable of type int, it might seem that

\[ \text{nextvalue } = 5 + \text{nextvalue } + 3 \]

could more compactly be written as

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

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

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

```cpp
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) evaluate a given arithmetic expression involving operands of types unsigned int
   and int only;
(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 representa-
   tion, for some b ∈ 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 7 Give the logical parenthesizations for the following expressions. (In order
   to avoid (misleading?) hints, we have removed the spaces that we usually include
   for the sake of better readability.)
   \[ \begin{align*}
   a) & \quad c=a?++\cdot b \quad b) \quad c=a+b \quad c) \quad c=a-b \\
   d) & \quad a\cdot a+b \quad e) \quad b++\cdot a+b \quad f) \quad a++\cdot b \\
   g) & \quad 7+a=b+2 \quad h) \quad a\cdot 3+\cdot b++\cdot a+ \quad i) \quad b++\cdot a
   \end{align*} \]

Exercise 8 Examine the expressions from Exercise 7 for validity. For an invalid
   expression, argue why it is invalid. For those expressions which are valid, provide
   a step-by-step evaluation, supposing that initially a has value 5, b has value 2, and
   the value of c is undefined.

Exercise 9 Compute by hand binary representations of the following decimal numbers.
   \[ \begin{align*}
   a) & \quad 15 \quad b) \quad 172 \quad c) \quad 329 \quad d) \quad 1022
   \end{align*} \]
2.3 Booleans

This section discusses the type bool used to represent Boolean values or
Booleans, for short. You will see a number of operations on Booleans
and why only a few of these operations suffice to express all the others.
You will learn how to evaluate expressions involving the type bool, using
short-circuit evaluation.

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

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

2.3.1 Boolean functions

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

Clearly the number of different Boolean functions is finite for every fixed \( n \); Exercise 14 asks you to show what exactly their number is. To give you a first hint: For
\( n = 1 \) there are only four Boolean functions, the two constant functions \( c_0 : x \mapsto 0 \) and \( c_1 : x \mapsto 1 \), the identity \( \text{id} : x \mapsto x \) and the negation \( \text{NOT} : x \mapsto \neg x \), where \( \mathcal{B} := \{0, 1\} \).

In the following we restrict our focus to binary Boolean functions, that is, functions
from \( \mathcal{B}^2 \) to \( \mathcal{B} \). As \( \mathcal{B}^2 \) consists of four elements only, such a function is most conveniently
described as a small table that lists the function values for all possible arguments. An
example for a binary Boolean function is \( \text{AND} : (x, y) \mapsto x \land y \) shown in Figure 4(a).
It is named AND because \( x \land y = 1 \) if and only if \( x = 1 \) and \( y = 1 \). You may guess
why the function \( f : (x, y) \mapsto x \lor y \) defined in Figure 4(b) is called OR. In fact, there
are two possible interpretations of the word “or”: You can read it as “at least one of”,
but just as well it can mean “either . . . or”, that is, “exactly one of”. The function that

Figure 4: Examples for binary Boolean functions.

Completeness. Figure 4 shows just a few examples. However, in a certain sense, it
shows you everything about binary Boolean functions. Some of these functions are
so fundamental that every binary Boolean function can be generated from them. For
example, consider the following function composed of AND and XOR: \( f : (x, y) \mapsto (x \land y) \oplus x \land y \).
(As you will show in Exercise 15, XOR is associative and hence there is
no need to further parenthesize the expression.) It is easily seen\(^{10}\) that \( f = x \lor y \), that
is, OR can be expressed as a composition of XOR and AND. Similarly, the function
\( \text{NAND} : (x, y) \mapsto x \uparrow y \) described in Figure 4(d) can be expressed as the composition
\( \text{NOT} \circ \text{AND} \) (hence the name . . . ). Let us formally define what we mean by “expressed
as a composition”.

Definition 2 Let \( \pi_1 : (x, y) \mapsto x \) and \( \pi_2 : (x, y) \mapsto y \) denote the binary projections, and
let \( c_0 : (x, y) \mapsto 0 \) and \( c_1 : (x, y) \mapsto 1 \) be the two constant functions.

Now consider a set \( \mathcal{F} \) of binary functions. A binary function \( f \) is called generated
by \( \mathcal{F} \) if \( f \in \mathcal{F} \) or \( \pi_1, \pi_2, c_0, c_1 \), or if there are binary functions \( f_1, f_2, f_3 \) that are generated
by \( \mathcal{F} \) such that

\[
f(x, y) = f_1(f_2(x, y), f_3(x, y)), \quad \forall x, y.
\]

For a set \( \mathcal{F} \) of binary functions, a set \( \mathcal{F} \) of binary functions is said to be complete
if and only if every function \( f \) in \( \mathcal{F} \) can be generated by \( \mathcal{F} \).

We have seen above that OR can be generated from \( \{\text{AND}, \text{XOR}\} \) as

\[
\text{XOR}(\text{AND}(x, y), \text{XOR}(x, y)).
\]

To employ the unary function NOT as a generator, we consider its “binaryisation”
\( \text{NOT}_2 : (x, y) \mapsto \neg x \) that ignores the second argument. We can then express
\( \text{NAND}(x, y) = \text{NOT}(\text{AND}(x, y)) \) formally as

\[
\text{NOT}_2(\text{AND}(x, y), \pi_2(x, y)),
\]

\(^{10}\)Just go through all four possible combinations of arguments.
Finally, we generate \( f_{0000} \) as
\[
f_{0000}(x, y) = c_0.
\]

Exercise 17 asks you to show that the sets \( \{\text{AND}, \text{NOT}\} \) \( \{\text{OR}, \text{NOT}\} \), and even the set that consists of the single function \( \text{NAND} \) are complete for the set of binary Boolean functions.

### 2.3.2 The type bool

In C++, Booleans are represented by the fundamental type `bool`. Its value range consists of the two elements `true` and `false` that are associated with the literals `true` and `false`, respectively. For example,
\[
\text{bool } b = \text{true};
\]
defines a variable `b` of type `bool` and initializes it to `true`.

Formally, the type `bool` is an integral type, defined to be less general than `int` (which in turn is less general than `unsigned int`, see Section 2.2.7).

Logical operators. The complete set of binary Boolean functions is available via the logical operators \( \&\& \) (AND), \( || \) (OR), and \( ! \) (NOT). Compared to the notation used in Section 2.3.1, we simply identify \( 1 \) with `true` and \( 0 \) with `false`. Both `\&\&` and `||` are binary operators, while `!` is unary. All operands are values of type `bool`, and all logical operators also return values of type `bool`. Like in logic, `\&\&` binds more strongly than `||`, and `!` binds more strongly than `\&\&`.\(^{11}\)

Relational operators. There is also a number of operators on arithmetic types whose result is of type `bool`. For each arithmetic type there exist the six relational operators `\(<\)`, `\(\leq\)`, `\(\geq\)`, `\(\text{==}\)` and `\(!=\)`. These are binary operators whose two value operands are of some arithmetic type and whose result is a value of type `bool`. The operators `\(<\)` and `\(\geq\)` correspond to the mathematical relations `\(<\)` and `\(\geq\)`, respectively. The operator `\(\text{==}\)` tests for equality and `\(!=\)` tests for inequality.

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

\[\text{Watch out! A frequent beginner's mistake is to use the assignment operator } = \text{ where the equality operator == is meant.}\]

As a general rule, logical operators have lower precedence than relational operators which in turn have lower precedence than arithmetic operators. For example,
\[
7 + x < y \&\& y \geq 3 \Rightarrow 3 + x
\]

\(^{11}\)Recall that an operator binds more strongly than another if its has higher precedence.
is logically parenthesized as

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

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

\(a = b = c\)

is not equivalent to

\(a = b \&\& b = c\).

By left associativity of \(\&\&\), the expression \(a = b = c\) is logically parenthesized as

\((a = b) = c\), if all of \(a, b,\) and \(c\) are variables of type int with value 0, the evaluation yields

\((0 == 0) = 0 \rightarrow true == 0 \rightarrow 1 == 0 \rightarrow false,\)

just the opposite of what you usually mean by \(a = b = c\).

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

\(! (x \& \& y) = (\neg x || \neg y)\)

and

\(! (x || y) = (\neg x \& \& \neg y)\).

These formulae can often be used to transform a Boolean expression (an expression of type bool) into a "simpler" equivalent form. For example,

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

can equivalently be written as

\(x >= y \&\& x + 1 <= z \&\& y > 5 * z \&\& y > 7 * z\)

which is clearly preferable in terms of readability.

For more details about precedences and associativities of the logical and relational operators, see Table 2. You may find this information helpful in order to solve Exercise 19.

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

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

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

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

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

```c
bool b = 5; // b is initialized to true
int i = b; // i is initialized to 1
```

2.3.3 Short circuit evaluation

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

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

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

\(x != 0 \&\& z / x > y\)
and be sure that this expression is always valid. If the right operand was evaluated for \( x = 0 \), then the result would be undefined.

2.3.4 Details

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

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

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

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

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

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

Then you can check that \( \phi_{\text{OR}}(4, 13) = 13, \phi_{\text{AND}}(13, 9) = 6, \phi_{\text{XOR}}(2) = 13 \).

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

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

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

<table>
<thead>
<tr>
<th>Description</th>
<th>Operator</th>
<th>Arity</th>
<th>Prec.</th>
<th>Assoc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitwise complement</td>
<td>( \neg )</td>
<td>1</td>
<td>16</td>
<td>right</td>
</tr>
<tr>
<td>bitwise and</td>
<td>&amp;</td>
<td>2</td>
<td>9</td>
<td>left</td>
</tr>
<tr>
<td>bitwise xor</td>
<td>( \oplus )</td>
<td>2</td>
<td>8</td>
<td>left</td>
</tr>
<tr>
<td>bitwise or</td>
<td>|</td>
<td>2</td>
<td>7</td>
<td>left</td>
</tr>
<tr>
<td>and assignment</td>
<td>( k )</td>
<td>2</td>
<td>4</td>
<td>right</td>
</tr>
<tr>
<td>xor assignment</td>
<td>( \oplus )</td>
<td>2</td>
<td>4</td>
<td>right</td>
</tr>
<tr>
<td>or assignment</td>
<td>|</td>
<td>2</td>
<td>4</td>
<td>right</td>
</tr>
</tbody>
</table>

Table 3: Precedence and associativity of bitwise operators.

2.3.5 Goals

Dispositional. At this point, you should ...

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

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

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

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

2.3.6 Exercises

Exercise 14 For \( n \in \mathbb{N} \), how many different Boolean functions \( f : B^n \rightarrow B \) exist? (G1)

Exercise 15 Prove or disprove that for all \( x, y, z \in B \)

\( a) \ (x \oplus y) \oplus z = x \oplus (y \oplus z) \) (i.e., XOR is associative)
\( b) \ (x \land y) \lor z = (x \lor y) \land (y \lor z) \) (i.e., (AND, OR) is distributive)

\[ \text{Having the equality operator, we can now use this as a shortcut for } \neg x = 0^* \]
Exercise 22 Find the logical parentheses in lines 9 and 10 of the following program. What can you say about the output of the following program? Characterize it depending on the input and explain your reasoning.

```cpp
#include <iostream>

int main() {
    int a;
    std::cin >> a;
    std::cout << (a++ < 3) << ".
"
    unsigned int b = a;
    b /= 2 + b / 2;
    std::cout << b << " \n";
    bool c = a < 1 || b != 0 && 2 * a / (a - 1) > 2;
    std::cout << c << " \n";
    return 0;
}
```