## 2.3 Booleans

*The truth always lies somewhere else.*

*Unknown*

*This section discusses the type* `bool` *used to represent truth values or Booleans, for short. You will see a number of operations on Booleans and why only 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 $f : \mathcal{B}^n \to \mathcal{B}$, 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 18 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 $id : x \mapsto x$ and the negation $NOT : x \mapsto \overline{x}$, where $\overline{0} := 1$ and $\overline{1} := 0$.

In the following we restrict our focus to unary and binary Boolean functions, that is, functions from $\mathcal{B}$ or $\mathcal{B}^2$ to $\mathcal{B}$. Such functions are most conveniently described as a small table that lists the function values for all possible arguments. An example for a binary Boolean function is $AND : (x, y) \mapsto x \wedge y$ shown in Figure 4(a). It is named AND because $x \wedge y = 1$ if and only if $x = 1$ *and* $y = 1$. You may guess why the



function $f : (x, y) \mapsto x \vee y$ 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 corresponds to the latter interpretation is shown in Figure 4(c). It is usually referred to as $XOR : (x, y) \mapsto x \oplus y$ or *exclusive or*. Figure 4(e) depicts the table for the unary function NOT.

| x | y | x ∧ y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

(a) AND.

| x | y | x ∨ y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

(b) OR.

| x | y | x ⊕ y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

(c) XOR.

| x | y | x ↑ y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

(d) NAND.

| x | $\overline{x}$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

(e) NOT.

**Figure 4:** *Examples for 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, XOR can be generated from AND, OR and NOT:

$$XOR(x, y) = AND(OR(x, y), NOT(AND(x, y))).$$

Informally, "either or" means "or" but not "and". Formulas like this are easily checked by going through all (four) possible combinations of arguments.

Similarly, the function $NAND : (x, y) \mapsto x \uparrow y$ described in Figure 4(d) can be generated from NOT and AND (hence the name ...):

$$NAND(x, y) = NOT(AND(x, y)).$$

Let us define what we mean by "generate".

**Definition 2** *Consider a set* $F$ *of boolean functions. A binary boolean function* $f$ *is called* **generated** *by* $F$ *if* $f$ *can be expressed by a formula that only contains the variables* $x$ *and* $y$*, the constants* 0 *and* 1*, and the functions from* $F$*.*

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

We are now prepared for a completeness proof.

**Theorem 1** *The set of functions* $\{AND, OR, NOT\}$ *is complete for the set of binary Boolean functions.*

**Proof.** Any binary Boolean function f is completely described by its *characteristic vector* $(f(0,0), f(0,1), f(1,0), f(1,1))$. For example, AND has characteristic vector $(0,0,0,1)$, or 0001 for short. Let $f_{b_1 b_2 b_3 b_4}$ denote the Boolean function with characteristic vector $b_1b_2b_3b_4$. For example, $\text{AND} = f_{0001}$.

In the first step of the proof, we show that all those functions can be generated whose characteristic vector contains a single 1. Indeed,

$$\begin{aligned} f_{0001}(x,y) &= \text{AND}(x,y), \\ f_{0010}(x,y) &= \text{AND}(x,\text{NOT}(y)), \\ f_{0100}(x,y) &= \text{AND}(y,\text{NOT}(x)), \\ f_{1000}(x,y) &= \text{NOT}(\text{OR}(x,y)). \end{aligned}$$

To check the formula for $f_{0010}$, for example, we can create a table for the function $\text{AND}(x,\text{NOT}(y))$ as in Figure 4 and convince ourselves that the resulting characteristic vector is 0010.

In the second step, we show that any function whose characteristic vector is nonzero can be generated. This is done by combining the already generated "single-1" functions through OR, which simply adds up their 1's. For example,

$$\begin{aligned} f_{1100}(x,y) &= \text{OR}(f_{1000}(x,y), f_{0100}(x,y)), \\ f_{0111}(x,y) &= \text{OR}(\text{OR}(f_{0100}(x,y), f_{0010}(x,y)), f_{0001}(x,y)). \end{aligned}$$

We abstain from working this argument out formally, since we believe that you get its idea. Finally, we generate $f_{0000}$ as

$$f_{0000}(x,y) = 0.$$

□

Exercise 21 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,

```
bool b = 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 rvalues of type `bool`, and all logical operators also return rvalues of tpe `bool`. Like in logics, `&&` binds more strongly than `||`, and `!` binds more strongly than `&&`.[9]

**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* `<`, `>`, `<=`, `>=`, `==`, and `!=`. These are binary operators whose two rvalue operands are of some arithmetic type and whose result is an rvalue of type `bool`. The operators `<=` and `>=` correspond to the mathematical relations $\leq$ and $\geq$, respectively. The operator `==` 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*.

> **Watch out!** A frequent beginner's mistake is to use the assignment operator `=` where the equality operator `==` is meant.

As a general rule, arithmetic operators bind more strongly than relational ones, and these in turn bind more strongly than the logical operators.

> **Boolean Evaluation Rule:** Binary arithmetic operators have higher precedence than relational operators, and these have higher precedence than binary logical operators.

For example, the expression

```
7 + x < y && y != 3 * z
```

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` $\longrightarrow$ *true* `== 0` $\longrightarrow$ `1 == 0` $\longrightarrow$ *false*,

just the opposite of what you usually mean by $a = b = c$.

[9] Recall that an operator binds more strongly than another if its has higher precedence.

**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 logics. 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) == (!x || !y)
```

and

```
!(x || y) == (!x && !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 23.

| Description | Operator | Arity | Prec. | Assoc. |
|---|---|---|---|---|
| logical not | ! | 1 | 16 | right |
| less | < | 2 | 11 | left |
| greater | > | 2 | 11 | left |
| less or equal | <= | 2 | 11 | left |
| greater or equal | >= | 2 | 11 | left |
| equality | == | 2 | 10 | left |
| inequality | != | 2 | 10 | left |
| logical and | && | 2 | 6 | left |
| logical or | \|\| | 2 | 5 | left |

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

**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.



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.

```
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 nonzero 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,[10] then the result would be undefined.

### 2.3.4 Details

**Naming.** The XOR function is also frequently called *antivalence* and denoted by $\not\leftrightarrow$. 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.

[10] having the equality operator, we can now use this as a shortcut for "x is 0"

**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 unary Boolean function* $f : \{0,1\} \to \{0,1\}$ *the* **bitwise operator** $\varphi_f$ *corresponding to* $f$ *is defined as* $\varphi_f(x) = \sum_{i=0}^{b} f(a_i) 2^i$.

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

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 $\varphi_{\text{OR}}(4,13) = 13$, $\varphi_{\text{NAND}}(13,9) = 6$, and $\varphi_{\text{NOT}}(2) = 13$.

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

| Description | Operator | Arity | Prec. | Assoc. |
|---|---|---|---|---|
| bitwise complement | `~` | 1 | 16 | right |
| bitwise and | `&` | 2 | 9 | left |
| bitwise xor | `^` | 2 | 8 | left |
| bitwise or | `\|` | 2 | 7 | left |
| and assignment | `&=` | 2 | 4 | right |
| xor assignment | `^=` | 2 | 4 | right |
| or assignment | `\|=` | 2 | 4 | right |

**Table 3:** *Precedence and associativity of bitwise operators.*

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 `~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.



### 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 18** *For* $n \in \mathbb{N}$*, how many different Boolean functions* $f : \mathcal{B}^n \to \mathcal{B}$ *exist?* (G1)

**Exercise 19** *Prove or disprove that for all* $x, y, z \in \mathcal{B}$ (G1)

a) $(x \oplus y) \oplus z = x \oplus (y \oplus z)$. (i.e., XOR is associative)

b) $(x \wedge y) \vee z = (x \vee z) \wedge (y \vee z)$. (i.e., (AND, OR) is distributive)

c) $(x \vee y) \wedge z = (x \wedge z) \vee (y \wedge z)$. (i.e., (OR, AND) is distributive)

d) $(x \uparrow y) \uparrow z = x \uparrow (y \uparrow z)$. (i.e., NAND is associative)

**Exercise 20** *For* $x_1, \ldots, x_n$*,* $n \in \mathbb{N}$*, give a verbal description of* $x_1 \oplus x_2 \oplus \ldots \oplus x_n$ *in terms of the* $x_i$*,* $1 \leq i \leq n$. (G1)

**Exercise 21** *Show that the following sets of functions are complete for the set of binary Boolean functions.* (G2)

a) $\{\text{AND}, \text{NOT}\}$

b) {OR, NOT}

c) {NAND}

d) {NOR}, where NOR := NOT ∘ OR.

e) {XOR, AND}

You may use the fact that {AND, OR, NOT} is a complete set of binary Boolean functions.

**Exercise 22** *Suppose* `a`, `b`, *and* `c` *are all variables of type* `int`. *Find values for* `a`, `b`, *and* `c` *for which the expressions* `a < b < c` *and* `a < b && b < c` *yield different results.* (G3)

**Exercise 23** *Parenthesize the following expressions according to operator precedences and associativities.* (G3)

a) `x != 3 < 2 || y && -3 <= 4 - 2 * 3`

b) `z > 1 && ! x != 2 - 2 == 1 && y`

c) `3 * z > z || 1 / x != 0 && 3 + 4 >= 7`

**Exercise 24** *Evaluate the expressions given in Exercise 23 step-by-step, assuming that* `x`, `y`, *and* `z` *are all of type* `int` *with* `x==0`, `y==1`, *and* `z==2`. (G3)

**Exercise 25** *What can you say about the output of the following program? Characterize it depending on the input and explain your reasoning.* (G4)

```
1   #include <iostream>
2   int main()
3   {
4     int a;
5     std::cin >> a;
6     std::cout << (a++ < 3) << ".\n";
7     bool b = a * 3 > a + 4 && !(a >= 5);
8     std::cout << (!b || ++a > 4) << ".\n";
9     return 0;
10  }
```

**Exercise 26** *Find the logical parentheses in lines 9 and 12 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.* (G4)

```
1  #include <iostream>
2
3  int main ()
4  {
```



```
5    unsigned int a;
6    std::cin >> a;
7
8    unsigned int b = a;
9    b /= 2 + b / 2;
10   std::cout << b << "\n";
11
12   bool c = a < 1 || b != 0 && 2 * a / (a - 1) > 2;
13   std::cout << c << "\n";
14
15   return 0;
16 }
```

### 2.3.7 Challenges

**Exercise 27** *The* Reverse Polish Notation *(RPN) is a format of writing expressions without any parentheses. RPN became popular in the late nineteensixties when the company Hewlett-Packard started to use it as input fomat for expressions on their desktop and handheld calculators.*

*In RPN, we first write the operands, and then the operator (that's what the* Reverse *stands for). For example, the expression*

$$\mathrm{AND}(\mathrm{OR}(0,\mathrm{NOT}(\mathrm{AND}(0,1))),1)$$

*can be written like this in RPN:*

0 0 1 AND NOT OR 1 AND.

*The latter sequence of operands and operators defines a specific evaluation sequence of the expression, see Section 2.2.3. To evaluate an expression in RPN, we go through the sequence from left to right; whenever we find an operand, we don't do anything, but when we find an operator (of arity* $n$*), we evaluate it for the* $n$ *operands directly to the left of it and replace the involved* $n+1$ *sequence elements by the result of the evaluation. Then we go to the next sequence element. In case of our example above, this proceeds as follows (currently processed operator in bold):*

0 $\underbrace{0\ 1\ \textbf{AND}}_{0}$ NOT OR 1 AND

0 $\underbrace{0\ \textbf{NOT}}_{1}$ OR 1 AND

$\underbrace{0\ 1\ \textbf{OR}}_{1}$ 1 AND

$\underbrace{1\ 1\ \textbf{AND}}_{1}$

1

*To see that this is indeed a way of evaluating the original expression*

$$\mathrm{AND}(\mathrm{OR}(0,\mathrm{NOT}(\mathrm{AND}(0,1))),1),$$

*you can for example make a bottom-up drawing of an expression tree (Section 2.2.2) that corresponds to the evaluation sequence in RPN. You will find that this tree is also valid for the original expression.*

*Here comes the actual exercise. Write programs* `and.C`*,* `or.C`*, and* `not.C` *that receive as input a sequence* $s$ *of boolean values in* $\{0,1\}$ *("all operands to the left of the operator"). The output should be the sequence* $s'$ *that we get by replacing the last* $n$ *operands in* $s$ *with the result of evaluating the respective operator for them. In case of* `and.C` *and* `or.C`*, we use* $n=2$*, and for* `not.C` $n=1$*. For example, on input* $(1,1,0)$*, program* `and` *should output the sequence* $(1,0)$*, while* `not` *should yields* $(1,1,1)$*.*

*In addition, write programs* `zero.C` *and* `one.C` *that output the sequence* $s'$ *obtained by appending a* `0` *or* `1` *to the input* $s$*. Finally, write a program* `eval.C` *(with no input) that outputs the empty sequence.*

*The goal of all this is to evaluate boolean functions in RPN by simply calling the corresponding sequence of programs (preceded by a call to* `eval`*), where the output of one program is used as input for the next one in the sequence. In Unix and Linux this can elegantly be done via a* pipe*. For example, to evaluate the example expression from above in RPN, we simply type the command*

```
./eval |./zero |./zero |./one |./and |./not |./or |./one |./and
```

*This calls all listed programs in turn, where a separating pipe symbol* `|` *has the effect that the output of the program to the left of it is used as the input for ("is piped into") the program to the right of it.*

*Consequently, the whole aforementioned command should simply write* `1` *to standard output, the result of the evaluation. Also test your programs with some other RPN sequences, in particular the "obvious" ones of the form*

```
./eval |./zero |./one |./or
```

*(this one should output* `1`*) to make sure that they work as expected.*

**Hint**: *It is not necessary that your programs accept sequences* $s$ *of arbitrary length as input. A maximum length of 32, for example, is sufficient for all practical purposes.*