Solution to Exercise 23. A Boolean function has 2^n possible n-tuples of arguments, and the function is specified by assigning the value 0 or 1 to each of them. It follows that there are 2^{2^n} possibilities, and this is also the number of Boolean functions.

Solution to Exercise 24. This can be done by going through the truth tables.

a) XOR is associative.

x	y	z	$(x \oplus y) \oplus z$		$x \oplus (y \oplus z)$	
0	0	0	$0 \oplus 0 = 0$		$0 \oplus 0 = 0$	
0	0	1	$0 \oplus 1 = 1$		$0 \oplus 1 = 1$	
0	1	0	$1 \oplus 0 = 1$		$0 \oplus 1 = 1$	
0	1	1	$1 \oplus 1 = 0$		$0 \oplus 0 = 0$	
1	0	0	$1 \oplus 0 = 1$		$1 \oplus 0 = 1$	
1	0	1	$1 \oplus 1 = 0$		$1 \oplus 1 = 0$	
1	1	0	$0 \oplus 0 = 0$		$1 \oplus 1 = 0$	
1	1	1	$0 \oplus 1 = 1$		$1 \oplus 0 = 1$	

b) (AND, OR) is distributive.

x	y	z	$(x \wedge y) \vee z$		$(x \vee z) \wedge (y \vee z)$	
0	0	0	$0 \vee 0 = 0$		$0 \wedge 0 = 0$	
0	0	1	$0 \vee 1 = 1$		$1 \wedge 1 = 1$	
0	1	0	$0 \vee 0 = 0$		$0 \wedge 1 = 0$	
0	1	1	$0 \vee 1 = 1$		$1 \wedge 1 = 1$	
1	0	0	$0 \vee 0 = 0$		$1 \wedge 0 = 0$	
1	0	1	$0 \vee 1 = 1$		$1 \wedge 1 = 1$	
1	1	0	$1 \vee 0 = 1$		$1 \wedge 1 = 1$	
1	1	1	$1 \vee 1 = 1$		$1 \wedge 1 = 1$	

c) (OR, AND) is distributive.

x	y	z	$(x \vee y) \wedge z$		$(x \wedge z) \vee (y \wedge z)$	
0	0	0	$0 \wedge 0 = 0$		$0 \vee 0 = 0$	
0	0	1	$0 \wedge 1 = 0$		$0 \vee 0 = 0$	
0	1	0	$1 \wedge 0 = 0$		$0 \vee 0 = 0$	
0	1	1	$1 \wedge 1 = 1$		$0 \vee 1 = 1$	
1	0	0	$1 \wedge 0 = 0$		$0 \vee 0 = 0$	
1	0	1	$1 \wedge 1 = 1$		$1 \vee 0 = 1$	
1	1	0	$1 \wedge 0 = 0$		$0 \vee 0 = 0$	
1	1	1	$1 \wedge 1 = 1$		$1 \vee 1 = 1$	

d) NAND is not associative, since for example $(0 \uparrow 0) \uparrow 1 = 1 \uparrow 1 = 0$, but $0 \uparrow (0 \uparrow 1) = 0 \uparrow 1 = 1$.

Solution to Exercise 25. $x_1 \oplus x_2 \oplus \cdots \oplus x_n$ is true if and only if an *odd* number of the x_i are 1.

Solution to Exercise 26.

a) We only need to show that $\{\text{AND}, \text{NOT}\}$ generates OR, and then use the completeness of $\{\text{AND}, \text{OR}, \text{NOT}\}$. This is simply DeMorgan's Law: we have

$$\text{OR}(x, y) = \text{NOT}(\text{AND}(\text{NOT}(x), \text{NOT}(y))).$$

b) This is similar; we only need to express AND using DeMorgan's Law:

$$\text{AND}(x, y) = \text{NOT}(\text{OR}(\text{NOT}(x), \text{NOT}(y))).$$

c) If we can generate NOT from NAND, we are done, since NOT and NAND can obviously generate AND, and then we use a). Indeed,

$$\text{NOT}(x) = \text{NAND}(x, x).$$

d) As in c), we are done once we can generate NOT from NOR, since we can then get OR and use b). Indeed,

$$\text{NOT}(x) = \text{NOR}(x, x).$$

e) Once we are able to generate NOT, we are done, since we can use a). But this is easy:

$$\text{NOT}(x) = \text{XOR}(x, 1).$$

Solution to Exercise 27. Values that work are -1 for `a`, 0 for `b` and 1 for `c`. Since `a < b < c` is parenthesized as `(a < b) < c` and `a< b` evaluates to *true*, the latter value is implicitly converted to the integer value 1. The comparison `1 < 1` then yields value *false*. The expression `a < b && b < c`, on the other hand, yields value *true*.

Solution to Exercise 28.

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))`

Solution to Exercise 29.

a) `(x != (3 < 2)) || (y && ((-3) <= (4 - (2 * 3))))` $\rightarrow$
`(0 != (3 < 2)) || (y && ((-3) <= (4 - (2 * 3))))` $\rightarrow$
`(0 != false) || (y && ((-3) <= (4 - (2 * 3))))` $\rightarrow$
`(0 != 0) || (y && ((-3) <= (4 - (2 * 3))))` $\rightarrow$
`false || (y && ((-3) <= (4 - (2 * 3))))` $\rightarrow$
`false || (true && ((-3) <= (4 - (2 * 3))))` $\rightarrow$
`false || (true && (`-3` <= (4 - (2 * 3))))` $\rightarrow$
`false || (true && (`-3` <= (4 - 6)))` $\rightarrow$
`false || (true && (`-3` <= `-2`))` $\rightarrow$
`false || (true && true)` $\rightarrow$
`false || true` $\rightarrow$
`true`

b) `((z > 1) && (((! x) != (2 - 2)) == 1)) && y` $\rightarrow$
`((2 > 1) && (((! x) != (2 - 2)) == 1)) && y` $\rightarrow$
`(true && (((! x) != (2 - 2)) == 1)) && y` $\rightarrow$
`(true && (((! 0) != (2 - 2)) == 1)) && y` $\rightarrow$
`(true && ((1 != (2 - 2)) == 1)) && y` $\rightarrow$
`(true && ((1 != 0) == 1)) && y` $\rightarrow$
`(true && (true == 1)) && y` $\rightarrow$
`(true && (1 == 1)) && y` $\rightarrow$
`(true && true) && y` $\rightarrow$
`(true && true) && 1` $\rightarrow$
`(true && true) && true` $\rightarrow$
`true && true` $\rightarrow$
`true`

c) `((3 * z) > z) || (((1 / x) != 0) && ((3 + 4) >= 7))` $\rightarrow$
`((3 * 2) > 2) || (((1 / x) != 0) && ((3 + 4) >= 7))` $\rightarrow$
`(6 > 2) || (((1 / x) != 0) && ((3 + 4) >= 7))` $\rightarrow$
`true || (((1 / x) != 0) && ((3 + 4) >= 7))` $\rightarrow$
`true`

Solution to Exercise 30. The output from line 6 is 1 (*true*) if and only if the input number n that is read in line 5 is less than 3. The expression `!b` is by de Morgan equivalent to

```
a * 3 <= a + 4 || a >= 5
```

which in turn is equivalent to `a <= 2 || a >= 5`. Observe that the variable `a` is incremented by one in line 6; therefore the expression `!b` in line 8 is *true*, unless $n \in \{2, 3\}$. Furthermore, for $n = 3$ the expression `++a > 4` in line 8 evaluates to 5>4, that is, *true*. Thus, the output from line 8 is 1 (*true*) if and only if $n \neq 2$.

Solution to Exercise 31. Here are the logical parentheses:

```
b /= (2 + (b / 2))
```

```
(a < 1) || ((b != 0) && (((2 * a) / (a - 1)) > 2))
```

Note that $2 + b \operatorname{div} 2 > b/2$, so `b` will be replaced by 1 at most. In fact, it is almost always replaced by 1, except for `a <= 2`. The first output is therefore 0 if `a <= 2` and 1, otherwise.

The second expression is definitely *true* for `a == 0`. For `a >= 1`, the value of the expression is determined by

```
(b != 0) && (((2 * a) / (a - 1)) > 2)
```

For `a <= 2`, we have `b == 0`, meaning that the value is *false*, and short circuit evaluation ensures that we don't divide by a non-positive number in this case.

For `a > 2`, we have `b == 1`, and the value of the expression is determined by

```
((2 * a) / (a - 1)) > 2
```

This yields *true* exactly for `a <= 3`. To summarize, the second output is 1 if and only if `a` has value 0 or 3.

Solution to Exercise 32. We can model the sequence s for example by a variable of type `unsigned int`. The binary representation of its value then yields the sequence of bits. To append a bit in the end, we simply multiply by two and add the bit. To remove bits in the end, we (repeatedly) divide by two. The last bit can at any stage be obtained through the modulus operator. To realize the replacement of bits in s, we use these mechanisms, along with the implicit conversions from `bool` to `unsigned int` and vice versa.

The empty sequence can be represented by value 0, but if you think about it, this value also represents *any* sequence of zeros. This means that the following programs can't issue an error message if the sequence of operands is too short for the current operator: logically, the sequence is filled up with zeros until "$-\infty$".

We could alternatively say that we only use `unsigned int` values whose binary representation starts with a 1 (and then value 1 would correspond to the empty sequence). However, at this stage we have no way of testing in a C++ program whether a number equals 1, so we can't issue an error message anyway.

Here are the six required programs.

```
1  // Program: and.C
2  // Replace last two bits in a sequence of bits by their AND
3
4  #include<iostream>
5
6  int main()
7  {
8    // input sequence: the binary representation of s
9    unsigned int s;
10   std::cin >> s;
11
12   bool x = s / 2 % 2; // second-to-last bit
13   bool y = s % 2; // last bit
```

```
14    std::cout << 2 * (s / 4)  + (x && y); //  x, y -> AND(x,y)
15    return 0;
16  }
```

```
1   // Program: or.C
2   // Replace last two bits in a sequence of bits by their OR
3
4   #include<iostream>
5
6   int main()
7   {
8     // input sequence: the binary representation of s
9     unsigned int s;
10    std::cin >> s;
11
12    bool x = s / 2 % 2; // second-to-last bit
13    bool y = s % 2; // last bit
14    std::cout << 2 * (s / 4)  + (x || y); //  x, y -> OR(x,y)
15    return 0;
16  }
```

```
1   // Program: not.C
2   // Flip last bit in a sequence of bits
3
4   #include<iostream>
5
6   int main()
7   {
8     // input sequence: the binary representation of s
9     unsigned int s;
10    std::cin >> s;
11
12    bool x = s % 2; // last bit
13    std::cout << 2 * (s / 2) + !x; // x -> NOT(x)
14    return 0;
15  }
```

```
1   // Program: zero.C
2   // Append 0 to a sequence of bits
3
4   #include<iostream>
5
6   int main()
7   {
8     // input sequence: the binary representation of s
9     unsigned int s;
10    std::cin >> s;
11
12    std::cout << 2 * s; // append 0
13    return 0;
14  }
```

```
1   // Program: one.C
2   // Append 1 to a sequence of bits
3
4   #include<iostream>
5
6   int main()
```

```
 7  {
 8    // input sequence: the binary representation of s
 9    unsigned int s;
10    std::cin >> s;
11
12    std::cout << 2 * s + 1; // append 1
13    return 0;
14  }
```

```
 1  // Program: eval.C
 2  // Output an empty sequence of bits
 3
 4  #include<iostream>
 5
 6  int main()
 7  {
 8    // behaves like an infinite sequence of 0's
 9    std::cout << 0;
10    return 0;
11  }
```