Contents

Booleans

Variables

We can store if something is true or false ("The user has reached the age of majority", "The switch is on", "The user is using Windows", "This computer's clock indicates that we are in the afternoon", ...) in a variable of type boolean, which is also known as a boolean *flag*. Note that `true` and `false` are the only possible two values for boolean variables: there is no third option!

We can declare, assign, initialize and display a boolean variable (flag) as with any other variable:

```
bool learning_how_to_program = true;
Console.WriteLine(learning_how_to_program);
```

Operations on Boolean Values

Boolean variables have only two possible values (`true` and `false`), but we can use three operations to construct more complex booleans:

1. "and" (&&, conjunction),
2. "or" (||, disjunction),
3. "not" (!, negation).

Each has the precise meaning described here:

1. the condition "A and B" is true if and only if A is true, and B is true,
2. "A or B" is false if and only if A is false, and B is false (that is, it takes only one to make their disjunction true),
3. "not A" is true if and only if A is false (that is, "not" "flips" the value it is applied to).

The expected results of these operations can be displayed in *truth tables*, as follows:

Operation	Value
`true && true`	`true`
`true && false`	`false`
`false && true`	`false`
`false && false`	`false`

Operation	Value
`true || true`	`true`
`true || false`	`true`
`false || true`	`true`
`false || false`	`false`

Operation	Value
`!true`	`false`
`!false`	`true`

These tables can also be written in 2-dimensions, as can be seen for conjunction on wikipedia[1].

Equality and Relational Operators

Boolean values can also be set through expressions, or tests, that "evaluate" a condition or series of conditions as `true` or `false`. For instance, you can write an expression meaning "variable `myAge` has the value 12" which will evaluate to `true` if the value of `myAge` is indeed 12, and to `false` otherwise. *To ease your understanding*, we will write "expression $\rightarrow$ `true`" to indicate that "expression" evaluates to `true` below, but this is *not* part of C#'s syntax.

Here we use two kinds of operators:

- Equality operators test if two values (literal or variable) are the same. This works on all datatypes.
- Relational operators test if a value (literal or variable) is greater or smaller (strictly or largely) than an other value or variable.

Relational operators will be primarily used for numerical values.

[1] https://www.wikiwand.com/en/Truth_table#Logical_conjunction_(AND)

Equality Operators

In C#, we can test for equality and inequality using two operators, == and !=.

Mathematical Notation	C# Notation	Example
$=$	==	`3 == 4` $\rightarrow$ `false`
$\neq$	!=	`3!=4` $\rightarrow$ `true`

Note that testing for equality uses *two equal signs*: C# already uses a single equal sign for assignments (e.g. `myAge = 12;`), so it had to pick another notation! It is fairly common across programing languages to use a single equal sign for assignments and double equal for comparisons.

Writing `a != b` ("`a` is not the same as `b`") is actually logically equivalent to writing `!(a == b)` ("it is not true that `a` is the same as `b`"), and both expressions are acceptable in C#.

We can test numerical values for equality, but actually any datatype can use those operators. Here are some examples for `int`, `string`, `char` and `bool`:

```
int myAge = 12;
string myName = "Thomas";
char myInitial = 'T';
bool cs_major = true;
Console.WriteLine("My age is 12: " + (myAge == 12));
Console.WriteLine("My name is Bob: " + (myName == "Bob"));
Console.WriteLine("My initial is Q: " + (myInitial == 'Q'));
Console.WriteLine("My major is Computer Science: " + cs_major);
```

This program will display

```
My age is 12: True
My name is Bob: False
My initial is Q: False
My major is Computer Science: True
```

Remember that C# is case-sensitive, and that applies to the equality operators as well: for C#, the string `Thomas` is not the same as the string `thomas`. This also holds for characters like `a` versus `A`.

```
Console.WriteLine("C# is case-sensitive for string comparison: " + ("thomas" != "Th
Console.WriteLine("C# is case-sensitive for character comparison: " + ('C' != 'c'))
Console.WriteLine("But C# does not care about 0 decimal values: " + (12.00 == 12));
```

This program will display:

```
C# is case-sensitive for string comparison: True
C# is case-sensitive for character comparison: True
But C# does not care about 0 decimal values: True
```

Relational Operators

We can test if a value or a variable is greater than another, using the following *relational* operators.

Mathematical Notation	C# Notation	Example
$>$	`>`	`3 > 4` $\rightarrow$ `false`
$<$	`<`	`3 < 4` $\rightarrow$ `true`
$\geq$ or $\geqslant$	`>=`	`3 >= 4` $\rightarrow$ `false`
$\leq$ or $\leqslant$	`<=`	`3 <= 4` $\rightarrow$ `true`

Relational operators can also compare `char`, but the order is a bit complex (you can find it explained, for instance, in this stack overflow answer[2]).

Precedence of Operators

All of the operators have a "precedence", which is the order in which they are evaluated. The precedence is as follows:

Operator	
`!`	is evaluated before
`*`, `/`, and `%`	which are evaluated before
`+` and `-`	which are evaluated before
`<`, `>`, `<=`, and `>=`	which are evaluated before
`==` and `!=`	which are evaluated before
`&&`	which is evaluated before
`||`	which comes last.

- Operators with higher precedence (higher in the table) are evaluated before operators with lower precedence (lower in the table). For instance, in an expression like `2*3+4`, `2*3` will have higher precedence than `3+4`, and thus be evaluated first: `2*3+4` is to be read as `(2*3)+4 = 6 + 4 = 10` and *not* as `2*(3+4) = 2*7 = 14`.

[2] https://stackoverflow.com/a/14967721/

- Operators on the same row have equal precedence and are evaluated in the order they appear, from left to right: in `1-2+3`, `1-2` will be evaluated before `2+3`: `1-2+3` is to be read as `(1-2)+3 = -1 + 3 = 2` and *not* as `1-(2+3) = 1-5 = 4`.
- Forgetting about precedence can lead to errors that can be hard to debug: for instance, an expression such as `! 4 == 2` will give the error

```
The `!' operator cannot be applied to operand of type
↪  `int'
```

Since `!` has a higher precedence than ==, C# first attempts to compute the result of `!4`, which corresponds to "not 4". As negation (`!`) is an operation that can be applied only to booleans, this expression does not make sense and C# reports an error. The expression can be rewritten to change the order of evaluation by using parentheses, e.g. you can write `!(4 == 2)`, which will correctly be evaluated to `true`.