C introduction

Operators

Richard Mörbitz, Manuel Thieme

Contents

Let's calc

Each time you type in a value or a variable, it gets evaluated as an expression. You can use operators to make new expressions out of existing ones.

```
3;                /* value 3 */
int a = 5;        /* value 5 */
int b = a + 3;    /* value 8 */
```

Got the idea?

Operators are for calculations, comparisons and more.

Back to School

- +, -, *, / as all of you should know
- % is the modulo (remainder) operator
- *, /, % get evaluated before +, -
- Operations in ( ) are of higher precedence

Something to try out and think about:

```
char a = 120;    /* 120 */
char b = 2 * a;          /* not 240 */
```

You see, it's not that easy

- Variables may overflow
- You shall not divide by zero
- Integer division differs from floating point division
- You can use operators between different data types
  - mixing different sizes
  - mixing integer and floating point variables

```c
int i1 = 42, i2 = 23;
short s = 13;
float f = 3.14;

i1 / i2;     /* results in 1, not a real division */
i1 + s;      /* int and short, result is int */
i1 / f;      /* result is float, actual division */
```

Syntactic sugar

As you have seen, you can use any expression on the right side of the assignment operator.
This expression often contains the variable it is assigned to.

To avoid redundancy, C offers the following short forms:

```
a += 4;         /* a = a + 4; */
a -= 4;         /* a = a - 4; */
a *= b;         /* a = a * b; */
b /= 42;        /* b = b / 42; */
b %= 2;         /* b = b % 2; */
c++;            /* c = c + 1; */
++c;            /* c = c + 1; */
c--;            /* c = c - 1; */
--c;            /* c = c - 1; */
```

The truth about expressions

Expressions can also be evaluated to truth values.
If a value or a variable equals 0, its corresponding truth value is *false*.
Otherwise it's *true*.
The representations of *true* and *false* are 0 and 1.
An expression containing relational operators gets evaluated to such a truth value.

Relational operators:

- $<$, $>$, $<=$, $>=$
- $==$ for "equal to"
- $!=$ for "not equal to"

Do not get confused

Imagine the following

```
(5 < 7) == 1;    /* evaluated to 1 */
```

Why?

Do not get confused

Imagine the following

```
(5 < 7) == 1;    /* evaluated to 1 */
```

Why?

- $(5 < 7)$ is **true** $\rightarrow$ **1**

Do not get confused

Imagine the following

```
(5 < 7) == 1;    /* evaluated to 1 */
```

Why?

- $(5 < 7)$ is **true** $\rightarrow$ **1**
- $1 == 1$ is **true** $\rightarrow$ **1**

A sign meant...

Assignments are expressions that get evaluated and have a truth value, too. Consider:

```
c = 0;                /* false */
c = 2 * 5;            /* 2 * 5 = 10, c = 10, true */
c = (0 < 1);          /* 0 < 1 = true, c = 1, true */

a = (b == (c = d));  /* Wat? */
```

$c{+}{+}$ expressions are evaluated before the increment while $++c$ increments first (the same applies on $c--$ and $--c$):

```
int c = 42;
int a = c++;    /* evaluates to c      a is 42  c is 43 */
int b = ++c;    /* evaluates to c + 1  b is 44  c is 44 */
```

Boolean arithmetic

Truth values can be connected by boolean operators resulting in a new truth value.

- && for AND (results in **1** if both operands are true, else **0**)
- || for OR (results in **1** if at least one operator is true, else **0**)
- ! for NOT (results in **1** if the operand is false, else **0**)

Precedence order:
! $>$ && $>$ ||

Seems logical

- How do you get NAND, NOR and XOR?

Seems logical

- How do you get NAND, NOR and XOR?

```
int a, b;
...
!(a && b);   /* NAND */
!(a || b);   /* NOR */
a != b;      /* XOR */
```

Exercises

- You are now able to solve tasks 03 and 04.