Operatoren in C/C++:

C/C++/Java hat eine riesige Anzahl von Operatoren (im Vergleich zu anderen Programmier-sprachen). Außer den üblichen arithmetischen Operatoren +, -, *, / (und % für Ganzzahlen) gibt es

Inkrement und Dekrement : `--, ++` (Prä- und Post-)

```
int i = 1, j = ++i;      // i hat den Wert 2, j hat den Wert 2
int i = 1, j = i++;      // i hat den Wert 2, j hat den Wert 1
int i = 1, j = --i;      // i hat den Wert 0, j hat den Wert 0
int i = 1, j = i--;      // i hat den Wert 0, j hat den Wert 1
```

Diese Operatoren erhöhen den Operanden um 1 bzw. erniedrigen ihn um 1. Daher ist es hier erforderlich, dass der Operand veränderbar ist (keine Konstante!) und einen festen Speicherplatz hat (also z.B. kein Zwischenergebnis einer Rechenoperation ist):

```
++ 1;       // Fehler 1 ist Konstante und kann nicht erhöht werden
++(x++);    // Fehler (x++) ist Rechenergebnis
```

C/C++ verwenden diese Operatoren sehr häufig. Niemand addiert oder subtrahiert 1, wenn man es mit diesen Operatoren machen darf. Diese Operatoren dürfen nicht auf Gleitkommatypen angewendet werden.

logische Operatoren: `!` oder `not` (logisches not), `||` oder `or` (logisches oder), `&&` oder `and` (logisches und), `==` (logische Gleichheit), `!=` (logische Ungleichheit), `<, >, <=, >=` (arithmetischer Vergleich)

z.B.
```
x != 1
a < 3 && a >= -7
a < 3 and a >= -7    (wie in Python geht es auch)
```

Das Ergebnis dieser Operatoren unterscheidet sich jedoch in den einzelnen Sprachen. C++ hat den Datentyp `bool` und obige Operatoren liefern als Ergebnis `true` oder `false` zurück. C hat keinen boolschen Datentyp, daher ist das Ergebnis eines solchen Operators immer der `int 1` für wahr und der `int 0` für false.

Wenn man sich unsicher über sie Reihenfolge ist, setzt man runde Klammern:

`(a > -7) && (a < 3)` : a im offenen Intervall (-7,3)

`(a <= -3) || (a > 3)` : a nicht im halboffenen Intervall (-3,3]

ACHTUNG:

`-7 <= a <= 3` **ist gültiges C/C++ (keine Fehlermeldung, möglicherweise eine Warnung), ABER IMMER WAHR (ALSO NICHT DAS, WAS MAN SICH ERWARTET).**

Erklärung: Aufgrund der Regeln für die Assoziativät gilt:
`-7 <= a <= 3` wird `zu (-7 <= a) <= 3` (<= ist linksassoziativ)

d.h. `(-7 <= a)` wird ausgewertet: Wert `true` für wahr oder Wert `false` für falsch. Dieser Wahrheitswert wird nun numerisch mit 3 verglichen und dazu in einen `int` umgewandelt (`true` in 1, `false` in 0). In beiden Fällen ist das Ergebnis dieses Vergleichs immer `true`.

Bitoperatoren: ~ (bitweises Komplement), `|` (bitweises oder), `&` (bitweises und),
^ (bitweises exclusiv-oder XOR), >> (bitweises Rechtsverschieben),
<< (bitweises Linksverschieben)
z.B. `~0, x >> 2, a | 3`

Hierbei werden die Bits im Ergebnis aus den einzelnen Bits der Operanden berechnet. Diese Operationen werden sehr häufig in der Computergrafik, beim Chiffrieren von Daten usw. eingesetzt.

Zuweisungsoperatoren: `=, +=, -=, *=, /=, %=, |=, &=, ^=, <<=, >>=`
z.B. `x += 3` (steht kurz für `x = x + 3`)
`x >>= 2` (dasselbe wie `x = x >> 2`)

Der Zuweisungsoperator = weist dem Ausdruck auf der linken Seite (meist eine Variable) den Wert des Ausdrucks auf der rechten Seite zu. Dabei wird automatisch eine Typkonvertierung vorgenommen, falls es nötig ist.

```
double x;
x = 7/4;
```

Alle 2 Sprachen (und die meisten anderen auch) berechnen zuerst die rechte Seite: `int/int` ergibt `int`, also ist das Ergebnis `1`. Dieser Wert wird nun in den Typ von `x` konvertiert und ergibt `1.0`, was zugewiesen wird.

```
int x;
x = 7.2/4;
```

Die rechte Seite ist `double/int -> double/double -> double` mit Wert `1.8`. Das wird in einen `int` gewandelt (ergibt `1` in C/C++ und eine Fehlermeldung in Java, da Java freiwillig immer nur in größere Datentypen umwandelt.)

```
int x{7.2/4};          // Fehler!!!
```

Bei der Initialisierung mittels `{}` sind keine verjüngenden (narrowing) Typumwandlungen erlaubt, d.h. man darf in keinen kleineren Typ umwandeln (`double -> int` geht nicht).

Adress-, Referenz- und Verweisoperatoren: &, *

Komma-Operator: ,
Funktions-Aufruf-Operator: ( )
Index-Operator: []
Ternärer Fragezeichen-Operator ?: *Bedingung **?** Wert_wenn_true **:** Wert_wenn_false*

```
int i = (j > 0 ? 5 : 3);   // i bekommt Wert 5 (bei j > 0 ) sonst 3
```

und noch einige mehr!! Viele dieser Symbole können, je nach Zusammenhang, 2 oder sogar noch mehr unterschiedliche Bedeutungen haben, z.B.:

> Minus - als unärer Operator (Vorzeichen) oder als binärer Operator (Subtraktion)
> Ampersand & ist Adressoperator, Referenzoperator und Bitoperator usw.

WICHTIG: Bei den **meisten** binären Operatoren ist es **nicht** geregelt, in welcher Reihenfolge die Operanden berechnet werden**. Es gilt also meistens nicht: Was zuerst steht, kommt zuerst (nur das „logische und", das „logische oder" und der Komma-Operator müssen sich an die vorgegebene Reihenfolge halten!):**

```
Ausdruck1  +  Ausdruck2
```

C++ darf hier mit *Ausdruck1* oder auch mit *Ausdruck2* beginnen. Das ist wichtig, wenn es Abhängigkeiten zwischen den Ausdrücken gibt:

```
n = 4;
(n++)  + n
```

Wird (n++) zuerst ausgewertet, so ist n danach 5 und das Resultat ist 4 + 5 = 9. Im anderen Fall ist das Ergebnis 4 + 4 = 8. Solcher Code führt in der Fachsprache zu **„Undefined Behavior",** d.h. beide Varianten sind legitim, das Resultat hängt vom Compiler/Optimierungsstufe/... ab und daher **sollte solcher Code NIE verwendet werden!**

Anweisungen

Anweisungen werden aus obigen Operatoren und Operanden zusammengesetzt und gehen bis zu einem Strichpunkt. **Anweisungen werden in der Reihenfolge ausgeführt, in der sie stehen**. Die eingebauten Vorrangsregeln (z.B. Punkt vor Strich) und Assoziationsregeln (automatische Klammernsetzung) führen dazu, dass man sehr viele Anweisungen bilden kann, die zwar syntaktisch richtig (es werden keine Vorschriften verletzt) aber inhaltlich völlig falsch sind. Java bricht öfters mit der C-Kompatibilität und würde deshalb einige dieser dubiosen Anweisungen als Syntaxfehler anzeigen.

Warnungsbeispiele:

```
if (-7 <= a <= 3)  (s.o.) wäre in Java ein Fehler, in C/C++ nicht
if (-7 <= a & y <= 3)      // & statt &&
if (-7 <= a, y <= 3)       // Komma-Operator
if (x = 7)// (auch das wäre in Java ein Fehler)
            // Zuweisung statt Abfrage auf Gleichheit: richtig if (x == 7)
            // entspricht dem Code x = 7; if (true)
my_function;     // ruft keine Funktion auf, richtig my_function();

 x = n/3 + (n++);           // n++ erhöht n um 1
            // welcher Wert wird aber beim n/3 genommen (alt oder neu?)
```

NOCH WICHTIGER:
Vermeide unleserliche Konstruktionen wie z.B. `x+++y;` (heißt das `(x++) + y` oder eben doch `x + (++y)`, was völlig unterschiedlich ist? C/C++/Java hat natürlich eine eindeutige Antwort darauf, die aber nicht unbedingt mit unserer Absicht übereinstimmen muss!

`*x++`: bedeutet `(*x)++` oder `*(x++)`?

Wer sich nicht sicher ist, sollte auf jeden Fall Klammern setzen. Und wer nicht weiß, was das überhaupt bedeutet, muss noch eine Menge lernen.