Wie werden arithmetische Operationen ausgeführt

Was ist `1 + 5` ? richtige Antwort unter Linux/Windows: `6`
Was ist `-1 - 1U` ? richtige Antwort unter Linux/Windows: `4294967294`
Ist `(a + b) + c` das gleiche wie `a + (b + c)`? (Assoziativitätsgesetz) : **nicht immer**

Um diese Fragen richtig zu beantworten, muss man 2 Dinge wissen:

1) Mit welchen Datentypen rechnet C/C++
2) Wann wird das Ergebnis einer Rechenoperation „falsch“

C/C++ hat einige sehr komplizierte Regeln zu 1), hier sind die wichtigsten für binäre Operationen (`+, -, *, /, %`):

1) Sind beide Operanden **Gleitkommatypen**, wird mit dem **größeren** der beiden Typen gerechnet (`long double > double > float`) und der andere Operand vorher konvertiert. Das Ergebnis hat ebenfalls diesen größeren Typ.
2) Ist **ein Operand ein Gleitkommatyp, der andere ein Ganzzahltyp**, so wird mit dem **Gleitkommatyp** gerechnet und der Ganzzahltyp vorher in den Gleitkommatyp konvertiert. Das Ergebnis hat diesen Gleitkommatyp.
3) **Sind beide Operanden Ganzzahltypen <= `int`, so wird mit `int` gerechnet** und (wenn nötig), beide Operanden zu `int` konvertiert. Das Ergebnis ist **`int`.**
4) Ist ein Operand ein **`unsigned int`**, und der andere Operandentyp <= `unsigned int`, wird mit `unsigned int` gerechnet und der andere Operand in `unsigned int` konvertiert. Das Ergebnis ist `unsigned int`. **Bei Vergleichen und bei manchen arithmetischen Operationen geht das oft schief!** Der Compiler warnt daher zu Recht, wenn ein `unsigned` mit einem `signed` Datentyp kombiniert wird!
5) Regeln wie 3) und 4) gelten sinngemäß, wenn man `int` durch `long` oder `long long` ersetzt und wenigstens ein Operand diesen Typ hat.

`double + float`	`-> double + double = double`	(Regel 1)
`int + float`	`-> float + float = float`	(Regel 2)
`int + int`	`-> int`	(Regel 3)
`char + short`	`-> int + int = int`	(Regel 3)
`unsigned + unsigned`	`-> unsigned`	(Regel 4)
`unsigned + char`	`-> unsigned + unsigned = unsigned` **(oder Katastrophe!)**	(Regel 4)
`long + int`	`-> long + long = long`	(Regel 5)

Überraschende Ergebnisse (finde eine Erklärung dafür und was ist der Ergebnistyp):

```
1)   -2   + 1U                          ->   4294967294
2)   -2LL + 1U                          ->   -1
3)   -2LL + 1ULL                        ->   18446744073709551615
4)   2'000'000'000  + 200'000'000       ->   -2094967296
5)   3'000'000'000  + 200'000'000       ->   3200000000
6)   2'000'000'000U + 2'000'000'000     ->   4000000000
7)   2'000'000'000U + 3'000'000'000     ->    705032704
8)   25e-9  +1                          ->   1.0000000250
9)   25e-9f +1                          ->   1.0000000000
```

Wann rechnet C/C++ „falsch“ und warum?

1) Bei einer **Ganzzahl-Rechnung mit `unsigned` Ergebnis:** Hier kann ein **Überlauf** auftreten, d.h. das richtige Resultat „passt nicht“ in den Ergebnistyp hinein. Dann **muss** der Compiler die „überstehenden“ oberen Bits (die wichtigen!) weglassen, das Ergebnis wird falsch: `6)`
2) Bei einer **Ganzzahl-Rechnung mit `signed` Ergebnis:** Hier kann ein **Überlauf** auftreten, d.h. das richtige Resultat „passt nicht“ in den Ergebnistyp hinein. Das ist **immer „Undefined Behavior“.,** d.h. das Programm könnte an dieser Stelle alles tun (abstürzen, piepsen oder falsch weiterrechnen). Meistens lässt der Compiler die „überstehenden“ oberen Bits (die wichtigen!) weg und das Ergebnis wird falsch: `4)` Dadurch können Summen von positiven Zahlen negativ werden!
3) Bei einer **Ganzzahl-Rechnung** mit der Kombination von `signed` mit `unsigned` Operanden: Muss der Compiler eine Umwandlung einer negativen `signed` Zahl in einen `unsigned` Datentyp vornehmen, so tritt hierbei immer ein Fehler auf: `1)` `3)`
4) Der Datentyp einer Ganzzahl-Konstanten ist nicht immer `int` sondern groß genug, um die Konstante aufzuzeichnen (falls es so einen Typ gibt!). Die Konstante `2'000'000'000` ist deshalb ein `int`, die Konstante `3'000'000'000` unter Linux ein `long` und unter Windows ein `long long`: `5)`.
5) Bei Gleitkomma-Rechnungen: Hier gibt es **keinen Überlauf** sondern sehr oft Rundungsfehler, d.h. das Ergebnis lässt sich nicht exakt mit den vorhandenen Mantissenbits darstellen und die CPU schneidet die überzähligen (hinteren) Bits ab: `8)`
   Kein Überlauf bedeutet, dass Summen und Produkte positiver Zahlen stets positiv sind etc. Allenfalls kann Unendlich herauskommen, wenn das Ergebnis zu groß für den Datentyp wird.

Zusammengesetzte Formeln, Ausdrücke (Expressions)

Diese werden in einzelne Rechenoperationen aufgeteilt. Dabei wird der Operator-Vorrang (der regelt die Reihenfolge von unterschiedlichen Operationen) und die Operator-Assoziativität (der regelt die Reihenfolge gleichwertiger Operationen) beachtet. Danach werden die **einzelnen Rechenschritte unabhängig** nach obigen Regeln behandelt. Eine explizite Klammerung durch den Programmierer hat immer Vorrang vor den eingebauten Regeln:

`(4 + 3) * 2` ist nicht dasselbe wie `4 + 3 * 2` (Punkt vor Strich)
`4 + 3 + 1` entspricht `(4 + 3) + 1` (+ ist links-assoziativ).

Eine Tabelle mit allen Operator-Wertigkeiten und -Assoziativitäten findet man in den Nachschlagewerken.

```
int + char + long -> (int + char) + long ->
     (int + int) + long ->
     int + long ->
     long + long ->
     long
```

Beachte hier, dass wirklich **JEDE** Operation einzeln betrachtet wird. Es könnte also durchaus bei der ersten Addition `(int + int)` ein Überlauf auftreten, der bei der durchgängigen Rechnung mit `long` nicht aufgetreten wäre. Dennoch wird diese erste Addition mit `int`-Daten berechnet und erst das (möglicherweise falsche übergelaufene) Ergebnis in `long` umgewandelt. Das **Assoziativgesetz** gilt für diese Arithmetik **NICHT**:

```
( 2'000'000'000U + 3'000'000'000U ) + 0LL    ->  705032704
2'000'000'000U + ( 3'000'000'000U + 0LL )    -> 5000000000
```