→ Rvalue and Lvalue

↳ Kann verwirrend sein, ist oft aber nicht schwer wenn man es sich vorsichtig überlegt.

→ L-value: lvalue simply means an object that has an identifiable location in memory

↳ wir werden später noch sehen was Addressen sind

→ R-value: r-value simply means, an object that has no identifiable location in memory (i.e. having an address).

⇒ Ein Objekt/eine Variable muss nicht immer das Gleiche sein, es hängt von der expression ab.

Aber ihr könnt z.B. nie einen R-value links einer Zuweisung (i.e. einfaches Gleichzeichen) haben.

(weil es keinen Ort im memory hat i.e. ihr könnt nichts dort reinschreiben.)

Expressions

→ Zu Übungszwecken lassen sich ziemlich schwere Bsp. konstruieren, ihr sollt et aber versuchen in eurem Code mögl. keine mehr-deutigen expressions zu haben.

→ Hier https://en.cppreference.com/w/cpp/language/operator_precedence findet ihr Präzedenz und Assoziativität aller Operatoren.

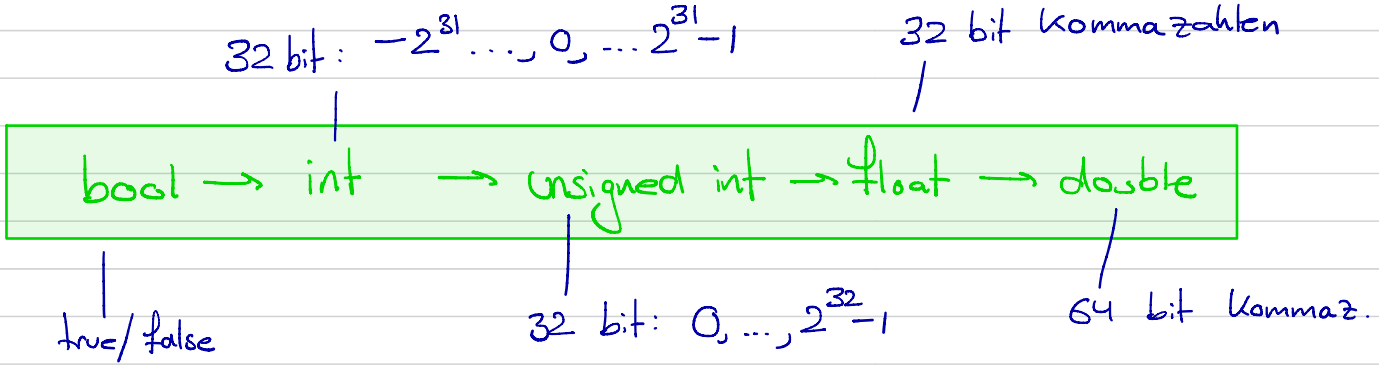
Alg. Vorgehen beim evaluieren

→ verschiedene Operatoren => Präzedenz, welche Operatoren müssen zuerst evaluiert werden? => Wo setze ich Klammern

→ gleiche Operatoren => Assoziativität => In welche Richtung wird evaluiert?

→ Operatoren versch. Datentypen => type conversion (implizite Konversion)

```
bool -> char -> short int -> int ->
unsigned int -> long -> unsigned ->
long long -> float -> double -> long double
```



Ein paar Bsp. (aus der Serie)

→ `a = b = 5;` → wir sehen = und = sind der gleiche Operator ⇒ Assoziativität! (in diesem Fall von rechts nach links)

⇒ `(a = (b = 5));`

jetzt können wir auswerten

```
a = b;
a = 5;
```

→ Pre vs. Post. increment

pre-increment `++i` "change then use"

e.g.
```
int a = 5;
int b = ++a;
// b ist 6
```
→ zuerst "change" dann "use" also zuerst a+1 rechnen und dann b zuweisen

post-increment `i++` "use then change"

e.g.
```
int a = 5;
int b = a++;
// b ist 5
```
zuerst "use" dann "change" also zuerst a (also 5) b zuweisen, dann a+1 rechnen

Boolean

→ true / false
→ false ≙ 0 , Zahl ≠ 0 entspricht true

→ Short Circuit evaluation:

true || .... → true
false && ... → false

wird vom Compiler garantiert, egal was ... ist

⇒ kann gewisse evaluationen effizienter machen.