Die `const` lvalue-Referenz-Übergabe (lv-reference to `const`)

```
void f(const int& x)      // x wird als lvalue-Referenz to const erstellt
// oder const int &x, const int & x, int const& x, int const &x
{
     x = 5;      // Fehler: x ist konstant
}
```

Es wird wieder eine lvalue-Referenz-Übergabe vereinbart, aber die Funktion garantiert dem Aufrufer, dass sie dieses Argument nicht verändern wird. C++ achtet genauestens darauf, dass das Argument weder in der Funktion selbst noch in von ihr aufgerufenen weiteren Funktionen geändert wird:

```
void g1(int x)            // kann Original-Argument nicht ändern (Wertübergabe)
{ ... }                   // darf aber x ändern (das ist ja nur die Kopie!)

void g2(const int& x)     // darf Argument x nicht ändern (const lv-Referenz)
{ ... }

void g3(int& x)           // g3 kann und darf x ändern (lv-Referenz)
{ ... }                   // ändert dadurch auch das Original-Argument

void f(const int& x)      // f darf x nicht ändern (const lv-Referenz)
{
     x = 5;          // Fehler: x darf nicht verändert werden
     g1(x);          // korrekt: x kann von g1 nicht verändert werden
     g2(x);          // korrekt: x darf von g2 nicht verändert werden
     g3(x);          // Fehler: x könnte von g3 geändert werden
}
```

Der Standard erlaubt die Benutzung des `const` nach der eigentlichen Typangabe:
`const int& a` ist dasselbe wie `int const& a`
Die meisten Programmierer schreiben aber `const` vor den Typ!

Was darf man per `const` lv-Referenz übergeben (am Beispiel: `int f(int&);`)
alles Umwandelbare

```
int x = 1; f(x);          // korrekt (Variable)
const int c = -1; f(c);   // korrekt (konstante Variable)
f(1);                     // korrekt (Konstante)
```

```
double y = 1.5; f(y);      // korrekt (umwandelbare Variable)
f(x + 1);                  // korrekt (Rechenergebnis)
f(abs(x));                 // korrekt (Rechenergebnis)
```

Erklärung: Wird eine Variable per lvalue-Referenz übergeben, muss diese im Speicher vorhanden sein und den richtigen Typ haben. C++ übergibt dann in Wahrheit die Adresse dieser Speicherstelle (=Referenz) an die Funktion. Es müssen daher für Konstante (diese haben keinen Speicherplatz), Rechenergebnisse und Argumente vom falschen Typ temporäre Kopien angelegt und benutzt werden, die nach dem Aufruf automatisch gelöscht werden. Der C++ Compiler erledigt alle diese Aufgaben automatisch! `f(1)` von oben ist zu folgendem Code äquivalent:

```
{     int tmp = 1;     // Konstante 1 in temporärer int-Variable speichern
      f(tmp);          // Aufruf mit Referenz auf die temporäre Variable
}                      // Variable tmp wieder löschen
```

`f(y)` von oben erzeugt das Code-Stück:

```
{     int tmp = y;     // im Speicher den konvertierten Wert ablegen
      f(tmp);          // Aufruf mit Referenz
}                      // Variable tmp wieder löschen
```

In beiden Fällen wird also in Wirklichkeit eine temporäre Variable an die Funktion per Referenz übergeben und nicht das Original. Die aufgerufene Funktion könnte daher das originale Argument gar nicht ändern (selbst wenn sie wollte), da sie nur eine Referenz auf das temporäre Objekt erhält. Das erlaubt C++ (auch **in allen alten Versionen)** nur, wenn die Funktion verspricht, das Original ohnehin nicht zu ändern, d.h. wenn eine konstante lv-Referenz übergeben wird.