## Datentypen

| Referenzen | **Alias** für bestehende Variable |
|---|---|

Referenzen können nur Variablen ihrens zugrundeliegenden Typs referenzieren. Sonst gibt es einen Fehler.

Ausserdem können Referenzen nur mit L-Werten initialisiert werden (also Werten mit einer Adresse im Speicher).

Funktionen, bei denen die Argumente Referenztyp haben, können ihre Aufrufargumente ändern. **Das ist eine sehr mächtige Anwendung von Referenzen.** Siehe beispielsweise die Funktion `shift` in `caesar_encrypt.cpp` aus der Vorlesung.

```
// Usage
int a = 3;
int& b = a; // reference to a
std::cout << b << "\n"; // Output: 3
a = 18;
std::cout << b << "\n"; // Output: 18
b = 25;
std::cout << a << "\n"; // Output: 25

// Issues
int& c = 3; // Error: 3 is not an lvalue (3 has no address)
bool d = false;
int& e = d; // Error: d is bool, e wants to reference an int
```

| Array | **"Massenvariable"** eines bestimmten Typs |
|---|---|

Wichtige Befehle:

**Definition:** `int my_arr[5] = {2, 3, 8, -1, 3};`
**Zugriff:** `my_arr[2] = 8 * my_arr[3];`
(siehe auch `[]`-Operator)

(Anstatt `int` gehen natürlich auch andere Typen.)
(Die Definition kann auch ohne Initialisierung erfolgen: `int my_arr[5];`)

Die Indizes beginnen bei 0.

( ... )

( ... )

Der Programmierer muss **selber** sicherstellen, dass die Indizes nicht über den Array hinausgehen.

Zuweisungen (ausser Initialisierung), Vergleiche, etc. müssen **elementweise** erfolgen.

Die Länge des Arrays **muss zum Kompilierzeitpunkt eindeutig bestimmbar sein.** (z.B. Literal oder `const`-Variable, die mittels Literal eingelesen wurde, etc.)

```
float a[10];

for (int i = 0; i < 10; ++i)
    a[i] = i; // a becomes {0 1 2 ... 9}

float b[10] = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
a = b; // NOT valid: array-copying is forbidden
       //     (should copy element-wise).
```

| Vektoren | **komfortabler Array** eines bestimmten Typs |
|---|---|

Erfordert: `#include <vector>`

Wichtige Befehle:

**Definition:** `std::vector<int> my_vec (length, init_value);`
**Zugriff:** (wie Array)

(Anstatt `int` gehen natürlich auch andere Typen.)
(Die Definition kann auch nur mit Längenangabe erfolgen:
`std::vector<int> my_vec (length)`)

**Unterschied zu Arrays:** Die Länge des Vektors muss **NICHT** zum Kompilierzeitpunkt eindeutig bestimmbar sein. Ausserdem besitzen Vektoren "Komfortfunktionen". (Mehr dazu später).

```
int len;
std::cin >> len; // Assume: len > 2

std::vector<int> my_vec (len, 0); // my_vec: 0, 0, 0, ..., 0
my_vec[1] = 3;                    // my_vec: 0, 3, 0, ..., 0
```

| `char` | Datentyp für **Zeichen** |
|---|---|

Literal: `'a'` für Zeichen (*einfache* Anführungszeichen)
Literal: `"Hello World"` für Strings (*doppelte* Anführungszeichen)

chars können sehr einfach zu `int` hin und her umgewandelt werden.

```
char c = 'd'; // default value 'd'
int i = c; // convert char --> int (here: 'd' --> 100)
++c;       // increase value of c to 101 (which is 'e')
++i;
std::cout << (c == i) << "\n"; // compare 101 == 101
std::cin >> c; // read single character from user
```

## Input/Output

| `std::noskipws` | **Whitespaces einlesen** |
|---|---|

Erfordert: `#include <ios>` oder `#include <iostream>`

```
char c;
// Version 1: Assume the user enters:
// a  b
std::cin >> c; // read 'a'
std::cin >> c; // read 'b'

// Version 2: Assume the user enters again:
// a  b
std::cin >> std::noskipws;
std::cin >> c; // read 'a'
std::cin >> c; // read ' '
std::cin >> c; // read ' '
std::cin >> c; // read 'b'
```

## Operatoren

| `my_array[...]` | **Array-** und **Vektor-Zugriff** (Subskript-Operator) |
|---|---|
| Präzedenz: 17 und Assoziativität: `links`<br><br>Nicht vergessen: Array-Indizes beginnen bei `0` und nicht `1` | |

```
int a[] = {8, 9, 10, 11};
std::cout << a[0]; // outputs 8
a[3] = 5; // a is 8, 9, 10, 5
```