Datentypen

`const` Referenzen	`const`-**Alias** für bestehende Variable

Im Prinzip funktionieren `const` Referenzen so wie normale Referenzen, bloss dass der Schreibzugriff auf das Ziel der Referenz *via diese Referenz* verboten ist.

Ein weiterer Unterschied ist, dass `const` Referenzen R-Werte beinhalten können. Dann wird jeweils ein temporärer Speicher für den R-Wert erstellt, der solange gültig ist, wie die `const` Referenz selbst. Dies erlaubt beispielsweise, eine Funktion bezüglich Call-by-Reference trotzdem mit R-Werten aufzurufen.

Zu beachten ist auch, dass man keine nicht-`const` Referenz mit einer `const` Referenz initialisieren darf.

```
double a = 3.0;
double& b = a; // non-const reference
const double& c = a; // const reference

c = 4.0; // Error: write-access forbidden
a = 5.0; // this works, a can be changed through itself
b = 6.0; // this works, a can be changed through non-const refs

std::cout << c << "\n"; // Output: 6.0, read-access is allowed.
double& d = c; // Error: non-const ref from const ref not allowed
const double& e = 5.0; // this works for const references.
```

`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)
neues Element hinten: `my_vec.push_back(5)`

(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" wie beispielsweise `push_back()`.

```
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
my_vec.push_back(5);              // my_vec: 0, 3, 0, ..., 0, 5
                                  //   [length increases by 1]
```

char	Datentyp für **Zeichen**

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

Ein String-Literal wird als Array des Typs `char` mit passender Länge gespeichert. Es wird immer mit '\0' terminiert (hat also ein Zeichen mehr als "Buchstaben").

```
char a[] = "my text"; // "my text" generates:   my text\0
a[3] = '5'; // Changes a to:    my 5ext\0

// output
for (int i = 0; a[i] != '\0'; ++i)
    std::cout << a[i] << "\n";
// Note: '\0' marks the end of the string.
```

Standardbibliothek

`std::swap`	**Tausche Werte** der Argumente.
Erfordert: `#include <utility>`	

```
int a = 3;
int b = 4;
std::swap(a, b);
std::cout << a << "\n"; // Output: 4
std::cout << b << "\n"; // Output: 3
```

Operatoren

`my_array[...]`	**Array-** und **Vektor-Zugriff** (Subskript-Operator)
Präzedenz: 17 und Assoziativität: `links`	

```
int a[] = {1, 2, 3, 4};
a[3] = 5; // a is 1, 2, 3, 5
```