Datentypen

`const ...`	***Schreib*zugriff** auf Variable **verbieten**
Gemeint ist natürlich der Schreibzugriff *nach* der Initialisierung. const gibt es auch für Referenzen, siehe unten.	

```
int a = 3;
const int b = 4;
a = 5;           // valid
b = 3;           // not valid since b is const
int c = -2 * b;  // valid since just WRITE-access to b is
                 //     forbidden by "const"
```

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 `swap` 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
```

`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.
```

Vektoren	**"Massenvariable"** eines bestimmten Typs

Erfordert: `#include<vector>`

Wichtige Befehle:

Definition: `std::vector<int> my_vec (length, init_value);`
Zugriff: `my_vec[2] = 8 * my_vec[3];`
neues Element hinten: `my_vec.push_back(5)`

(Anstatt `int` gehen natürlich auch andere Typen.)

```
int len;
std::cin >> len; // Assume here: 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. (Der resultierende `int`-Wert ist auf den meisten Plattformen eine entsprechende Zahl gemäss ASCII-Code, siehe Vorlesungshandout 7, Slide 45.)

```
char ch = 'd';
int i = ch; // convert char --> int (here: 'd' --> 100)
++ch;       // increase to 101 which is 'e'
++i;
std::cout << (ch == i) << "\n"; // compare 101 == 101

// Read single character from user:
std::cin >> ch;
```

Operatoren

`my_vec[...]`	**Vektor-Zugriff** (Subskript-Operator)
Präzedenz: 17 und Assoziativität: `links` Nicht vergessen: 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
```