Datentypen

`Array (mehrdim.)`	**mehrdimensionale "Massenvariable"** eines bestimmten Typs

Wichtige Befehle:

Definition: `int my_arr[2][3] = { {2, 1, 6}, {8, -1, 4} };`
Zugriff: `my_arr[1][1] = 8 * my_arr[0][2];`

(Anstatt `int` gehen natürlich auch andere Typen.)
(Die Definition kann auch ohne Initialisierung erfolgen: `int my_arr[2][3];`)

```
int my_arr[2][3] = { {2, 1, 6}, {8, -1, 4} };
my_arr[1][1] = 8 * my_arr[0][2];
    // my_arr becomes
    //     2,   1,  6
    //     8,  48,  4
```

`Vektoren (mehrdim.)`	**komfortabler mehrdimensionaler Array** eines bestimmten Typs

Erfordert: `#include <vector>`

Wichtige Befehle:

Definition:
```
std::vector<std::vector<int> >
  my_vec (n_rows, std::vector<int>(n_cols,
                                   init_value))
```
Zugriff: (wie mehrdim. Array)

(Anstatt `int` gehen natürlich auch andere Typen.)
(Die Definition kann auch ohne Initialisierung erfolgen:
`std::vector<std::vector<int> > my_vec (n_rows, std::vector<int>(n_cols)) )`

Beachte: `> >` muss mit Abstand geschrieben werden. Sonst gibt es Doppeldeutigkeiten mit dem Eingabe-Operator `>>` (siehe `std::cin`).

(...)

(...)

```
std::vector<std::vector<int> > my_vec (2, std::vector<int>(4, 0));
my_vec[1][2] = 3;
    // my_vec becomes
    //     0,  0,  0,  0
    //     0,  0,  3,  0
```

std::string	**komfortablerer** Datentyp für **Zeichen**

Erfordert: `#include <string>`

Vorteile gegenüber `char`-Arrays:

variable Länge:	`std::string my_str (n, 'a');` (`n` kann variabel sein)
Länge abfragen:	`my_str.length()`
vergleichbar:	`text1 == text2`
hintereinander hängen:	`text1 += text2`
bequemer Output:	`std::cout << my_str;`

```
std::string my_word (5, 'a'); // initialize my_word as aaaaa
std::string ref (5, 'z');
my_word += ref; // append ref to my_word.
                // Afterwards my_word:   aaaaazzzzz
                // Afterwards ref:       zzzzz
for (int i = 0; i < my_word.length(); ++i)
    std::cin >> my_word[i]; // read user input into our word
if (my_word == ref) // impossible since lengths differ (5 VS 10)
    std::cout << "never output\n";
std::cout << my_word << "\n"; // output whole string at once
```

Generell

`switch`	**Fallunterscheidung**

Wird ein `case` nicht mit einem `break` abgeschlossen, so werden die darunter liegenden `case`s auch noch ausgeführt bis ein `break` erreicht wird.

Es gibt auch den Spezialfall `default`. Sofern der Programmierer ihn einbaut, so wird dieser Fall ausgeführt, wenn vorher kein Fall zutreffend war.

Die einzelnen Unterscheidungswerte müssen Konstanten sein.

```
char door;
std::cout << "Behind which door (a, b, or c) is the prize? ";
std::cin >> door;
switch (door) {
case 'a':
    std::cout << "You won the prize!\n";
    break;
case 'b':
    std::cout << "Nice try... ";
case 'c':
    std::cout << "Nope, wrong door.\n";
    break;
default:
    std::cout << "This door does not exist.\n";
}

    // Input: a
    // Output: You won the prize!
    // Input: b
    // Output: Nice try... Nope, wrong door.
    // Input: c
    // Output: Nope, wrong door.
    // Input: d
    // Output: This door does not exist.
```

Input/Output

`std::noskipws`	**Whitespaces einlesen**

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

```
// Assume that both times the following text is entered:
//     aa bb cc dd

// Version 1:
for (char input; std::cin >> input; )
    std::cout << input; // Output: aabbccdd

// Version 2:
std::cin >> std::noskipws;
for (char input; std::cin >> input; )
    std::cout << input; // Output: aa bb cc dd
```

`leerer Eingabestrom`	Prüfe, ob **mehr Eingaben vorhanden** sind.

Dahinter steckt eine Konvertierung von `std::cin` zu `bool`:

`true:`	weitere Eingaben vorhanden
`false:`	keine Eingaben mehr vorhanden

Wir brauchen diese Abfrage meistens, um eine Schleife solange laufen zu lassen, wie weitere Eingaben vorhanden sind. (siehe Beispiel unten)

Erfolgt die Eingabe per Tastatur, so kann die Eingabe durch drücken von `[Ctrl]+[D]` beendet werden.

```
char input;
int length_of_text = 0;
while(std::cin >> input)
    ++length_of_text;
std::cout << length_of_text;
```

Turtle

Turtle Plots	**Zeichnen** von Geraden

Erfordert: `#include "turtle.cpp"`

Die Turtle kennt 7 Befehle:

`turtle::forward();`	gezeichneter Schritt vorwärts
`turtle::jump();`	nicht gezeichneter Schritt vorwärts
`turtle::left(my_angle);`	Drehung nach links um 45 Grad
`turtle::right(my_angle);`	Drehung nach rechts um 45 Grad
`turtle::save();`	Position *und Blickrichtung* merken
`turtle::restore();`	Position *und Blickrichtung* laden
`turtle::colorcycle();`	Aktuelle Stiftfarbe ändern

Zu beachten ist, dass die Turtle mehrere Positionen speichern kann (mittels `turtle::save()`. Sind mehrere Positionen gespeichert, so lädt `turtle::restore()` die neueste und entfernt diese Position dann aus der Merkliste (somit ist dann die vorher zweitneuste Position neu die neuste).

```
// Draw a triangle (see below)
turtle::forward();
turtle::left(120);
turtle::forward();
turtle::left(120);
turtle::forward();

// Move to neutral position
turtle::left(120); // horizontal viewing direction
turtle::jump(10); // move away from triangle (without drawing)

// Draw a letter T (see below)
turtle::forward();
turtle::save(); // memorize middle of letter T
turtle::forward();
turtle::restore(); // go back to middle of letter T
turtle::right(90);
turtle::forward(2); // The argument means: 2 steps forward
-----------------------------------------------------------------
```