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];`
Vor dem ersten Lesezugriff auf ein Element sollte aber unbedingt mindestens ein Schreibzugriff darauf erfolgt sein.)

```
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.)

```
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:

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
std::cout << my_word.length() << "\n"; // output: 10
my_word[3] = 'b';   // change my_word to aaabazzzzz
if (my_word == ref) // false
    std::cout << "not output\n";
std::cout << my_word << "\n"; // output whole string at once
```

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 'b'
```

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)

```
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.h"`

Die Turtle kennt 8 Befehle:

`turtle::forward():`	gezeichneter Schritt vorwärts
`turtle::jump():`	nicht gezeichneter Schritt vorwärts
`turtle::left(my_angle):`	Drehung nach links
`turtle::right(my_angle):`	Drehung nach rechts
`turtle::save():`	Position *und Blickrichtung* merken
`turtle::restore():`	Position *und Blickrichtung* laden
`turtle::colorcycle():`	Farbe wechseln
`turtle::colorcycle2(d):`	Farbe wechseln (eigene Abstufung `d`)

Die Turtle kann mehrere Positionen speichern (mittels `turtle::save()`). `turtle::restore()` lädt dann die Neueste und entfernt diese Position 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
-----------------------------------------------------------------
```

Zeiger

Zeiger (generell)	**Adresse** eines Objekts im Speicher

Wichtige Befehle:

Definition:	`int* ptr = address_of_type_int;`
(ohne Startwert:	`int* ptr = 0;`)
Zugriff auf Zeiger:	`ptr = otr_ptr // Pointer gets new object.`
Zugriff auf Objekt:	`*ptr = 5 // Object gets new value 5.`
Adresse auslesen:	`int* ptr_to_a = &a; // (a is int-variable)`
Vergleich:	`ptr == otr_ptr // Same object?`
	`ptr != otr_ptr // Different objects?`

(Anstatt `int` gehen natürlich auch andere Typen.)
(Eine `address_of_type_int` kann man durch einen anderen Zeiger oder auch mittels dem Adressoperator `&` erzeugen (siehe Beispiel unten).)

Der **Wert** des Zeigers ist die Speicheradresse des **Objekts**, auf das er zeigt. Will man also das Objekt via diesen Zeiger verändern, muss man zuerst "zu der Adresse gehen". Genau das macht der Dereferenz-Operator `*`.

Beispiel: (Gelte `int a = 5;`)

Wert von `a`:	`5`
Speicheradresse von `a`:	`0x28fef8`
Wert von `a_ptr`:	`0x28fef8`
Wert von `*a_ptr`:	`5`

Ein Zeiger kann immer nur auf den entsprechenden Typ zeigen.
(z.B. `int* ptr = &a;` Hier muss `a` Typ `int` haben.)

```
int a = 5;
int* a_ptr = &a; // a_ptr points to a
a_ptr = a;   // NOT valid (same as:  a_ptr = 5; )
             //     5 is NOT an address and NOT an array.
a_ptr = &a;  // valid
*a_ptr = 9;  // a obtains value 9
std::cout << a << "\n";      // Output: 9
std::cout << *a_ptr << "\n"; // Output: 9
```