Datentypen

`float`	Datentyp für **Zahlen mit Nachkommastellen** (32 Bit nach IEEE 754)

Literal: `ohne Exponent: 288.18f, mit Exponent: 0.28818e3f`

Der Modulo-Operator `%` existiert für `float` nicht.

```
float a = 288.18f;
float b = 0.28818e3f / a; // computations work as expected
float c;
std::cin >> c; // float user input
```

`double`	grösserer Datentyp für **Zahlen mit Nachkommastellen** (64 Bit nach IEEE 754)

Literal: `ohne Exponent: 288.18, mit Exponent: 0.28818e3`

Unterschied zu `float`: `double` ist genauer (grössere Präzision und grösseres Exponenten-Spektrum), braucht aber mehr Platz im Speicher (`float`: 32 Bit, `double`: 64 Bit).
Der Modulo-Operator `%` existiert für `double` nicht.

```
double a = 288.18;
double b = 0.28818e3 / a; // computations work as expected
double c;
std::cin >> c; // double user input
```

Schleifen

`while (...) {...}`	`while`-Schleife

```
// Compute number of binary digits for input > 0
unsigned int bin_digits = 0;
unsigned int input;
std::cin >> input;
assert(input > 0);

while (input > 0) {
    input /= 2;
    ++bin_digits;
}
```

`do {...} while (...);`	`do`-Schleife

Der Unterschied zur `while`-Schleife ist, dass der Rumpf der `do`-Schleife mindestens einmal ausgeführt wird.
Sie hat ein ";" am Schluss.

```
int input;
do {
    std::cout << "Enter negative number: ";
    std::cin >> input;
} while (input >= 0);
std::cout << "The input was: " << input << "\n";
```

`break`	Schleife **abbrechen**

```
double input;
int n;
std::cin >> input >> n;

// Divide input by n numbers
// Stop if 0 is entered.
for (int i = 0; i < n; ++i) {
    double k;
    std::cin >> k;
    if (k == 0)
        break; // go straight to Output
    input /= k;
}

// Output
std::cout << input << " remains\n";
```

`continue`	zur **nächsten Iteration** springen

Bei der `for`-Schleife wird das Inkrement noch ausgeführt.

```
double input;
int n;
std::cin >> input >> n;

// Divide input by n numbers
// Skip entered 0's.
for (int i = 0; i < n; ++i) {
    double k;
    std::cin >> k;
    if (k == 0)
        continue; // go straight to ++i
    input /= k;
}

// Output
std::cout << input << " remains\n";
```

Generell

{ ... }	**Block**

Blöcke spielen eine grosse Rolle, wenn es darum geht, wo im Programm eine Variable gültig ist. So ist eine Variable ab ihrer Deklaration bis hin zum Ende des Blocks, in dem sie definiert wurde potentiell gültig.

```
int main () {
    unsigned int a;
    std::cin >> a;
    if (a < 4) {
        std::cout << a << " "; // a exists in nested blocks
        int b = 18;
        std::cout << b << " "; // b exists here too
    } else {
        std::cout << b << " "; // Error: b not declared yet
        int b = 11;
    }
    std::cout << a << " "; // a still exists here
    std::cout << b << "\n"; // Error: b does not exist anymore
    return 0;
}
```