!Right-most number == 1!			
binär	decimal	octoal	hexidec
0bN	N	0N	0xN
0000000	0	000	00
0000001	1	001	01
0000010	2	002	02
0000011	3	003	03
0000100	4	004	04
0000101	5	005	05
0000110	6	006	06
0000111	7	007	07
0001000	8	010	08
0001001	9	011	09
0001010	10	012	0A
0001011	11	013	0B
0001100	12	014	0C
0001101	13	015	0D
0001110	14	016	0E
0001111	15	017	0F
0010000	16	020	10
0010001	17	021	11
0010010	18	022	12
0010011	19	023	13
0010100	20	024	14
0010101	21	025	15
0010110	22	026	16
0010111	23	027	17
0011000	24	030	18
0011001	25	031	19
0011010	26	032	1A
0011011	27	033	1B
0011100	28	034	1C
0011101	29	035	1D
0011110	30	036	1E
0011111	31	037	1F
0100000	32	040	20
0100001	33	041	21
0100010	34	042	22
0100011	35	043	23
0100100	36	044	24
0100101	37	045	25
0100110	38	046	26
0100111	39	047	27
0101000	40	050	28

!Right-most number == 1!			
binär	decimal	octoal	hexidec
0bN	N	0N	0xN
0101001	41	051	29
0101010	42	052	2A
0101011	43	053	2B
0101100	44	054	2C
0101101	45	055	2D
0101110	46	056	2E
0101111	47	057	2F
0110000	48	060	30
0110001	49	061	31
0110010	50	062	32
0110011	51	063	33
0110100	52	064	34
0110101	53	065	35
0110110	54	066	36
0110111	55	067	37
0111000	56	070	38
0111001	57	071	39
0111010	58	072	3A
0111011	59	073	3B
0111100	60	074	3C
0111101	61	075	3D
0111110	62	076	3E
0111111	63	077	3F
1000000	64	100	40

F*(β, p, e_min, e_max)
largest:
1.1...p....1 * β^(e_max)
smallest (= "precision"):
β^(e_min)
of representable numbers
(β-1)*(β^(p-1))*(e_max-e_min+1)
^ *2 if negative too
if "0" counts too +1 ^

2^n	n
0.0625	-4
0.125	-3
0.25	-2
0.5	-1
1	**0**
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024	10

```cpp
// C++ program to find LCM of two numbers
#include <iostream>

// Recursive function to return gcd of a and b // ggT
public: int gcd(int a, int b){
  if (a == 0)
    return b;
  return gcd(b % a, a);
}

// Function to return LCM of two numbers // kgV
int lcm(int a, int b)
{
  return (a*b)/gcd(a, b);
}
```

```cpp
#include <iostream>
// how to initialize matrices
int main(){

int matrix[2][5] = {  // mxn
  {1, 2, 3, 4, 5},    // don't forget
  {6, 7, 8, 9, 10}    // the ","
};

std::cout << matrix[1][3];
// returns 9! because 0, 1, 2,..

return 0;
}
```

```cpp
#include <iostream>
#include <vector>
// PRE: 0 <= end <= v.size(), and
//      0 < denominations[0] < denominations[1] < .. denominators[end-1]
//      describes a (potentially empty)
//      sequence of denominations
// POST: return value is the number of ways to partition amount
//       using denominations from denominations[0], ..., denominations[end-1]
unsigned int partitions(unsigned int amount,
                        const std::vector<unsigned int>& denominations,
                        unsigned int end) {
    // Only one way to hold nothing.
    if (amount == 0) return 1;

    // ways = ways_0 + ... + ways_{end-1}, where ways_i is the number
    // of ways to partition amount using denominations[i] as the largest
    // denomination
    unsigned int ways = 0;
    // Iterate over all choices of larger denominations.
    for (unsigned int i = 0; i != end; ++i) {
        // If amount too small, cannot be done for this particular
        // denominations [way_i = 0], neither for next denominations due to
        // iteration order and denominations being increasing.
        if (amount < denominations[i]) {
            break;
        }
        // If d_i = denominations[i] fits, way_i = number of partitions of the
        // form (d_i,X) with X being a partition of amount-d_i using
        // denominations[0], ..., denominations[i]
        ways += partitions(amount - denominations[i], denominations, i + 1);
    }
    return ways;
}
```

```cpp
// QUICKSORT
// POST: begin denotes the start of an allocated memory chunk
//       end denotes the past-the-end pointer of this memory chunk
//       the range contains the sequence of integer values read from input stream is
void input(std::istream& is, int*& begin, int*& end) {
    unsigned int size;
    is >> size;
    begin = new int[size];
    for (unsigned int i = 0; i != size; ++i) {
        is >> begin[i];
    }
    end = begin + size;
}
// PRE: begin points within an allocated memory chunk
//      end points within the same memory chunk, not before begin
// POST: the sequence of integer values in memory range [begin; end)
//       has been printed to os, separated by single spaces.
void output(std::ostream& os, const int* begin, const int* end) {
    // NOTE: solutions outputting extra spaces at the end are accepted.
    bool first = true;
    for (const int* it = begin; it != end; ++it) {
        if (!first) {
            os << ' ';
        } else {
            first = false;
        }
        os << *it;
    }
}

// POST: the values pointed to by *a and *b have been swapped.
void swap(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

// PRE: begin points within an allocated memory chunk
//      end point later (strictly) within the same memory chunk
// POST: elements in range [begin; end) are the same (likely in different order)
//      Resulting pointer p points within range [begin; end)
//      *p is the old value of *begin
//      Elements in range [begin; p) are < to *p
//      Elements in range (p; end) are >= to *p.
int* pivot(int* begin, int* end) {
    // p points to pivot, b initially points to to p+1.
    int* p = begin;
    int* b = begin + 1;
    int* e = end;
    while (b != e) {
        if (*b < *p) {
            swap(b++, p++);
        } else {
            swap(b, --e);
        }
    }
    return p;
}
// PRE: begin points within an allocated memory chunk
//      end points later within the same memory chunk
// POST: elements in range [begin; end) are the same, but sorted in increasing
//       order.
void quicksort(int* begin, int* end) {
    if (begin != end) {
        int* piv = pivot(begin, end);
        //sort the sub-ranges on the both sides of pivot
        quicksort(begin, piv);
        quicksort(piv + 1, end);
    }
}
```

Datentypen I

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.

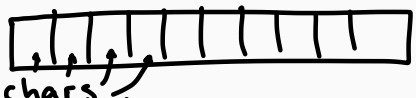
Zu beachten ist auch, dass man keine nicht-const Referenz mit einer `const` Referenz initialisieren darf.

```
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.
```

std::string

komfortablerer Datentyp für **Zeichen**

Erfordert: `#include<string>`



Vorteile:

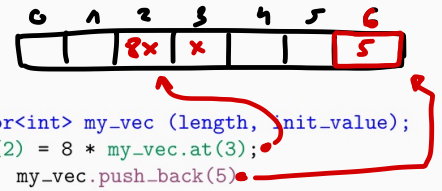
- **variable Länge:** `std::string my_str (n, 'a');` (n kann variabel sein)
- **Länge abfragen:** `my_str.length()`
- **vergleichbar:** `text1 == text2`
- **hintereinander hängen:** `text1 += text2`
- **bequemer Output:** `std::cout << my_str;`

```
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.at(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
```

Vektoren

"Massenvariable" eines bestimmten Typs

Erfordert: `#include<vector>`



Wichtige Befehle:

- **Definition:** `std::vector<int> my_vec (length, init_value);`
- **Zugriff:** `my_vec.at(2) = 8 * my_vec.at(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.at(1) = 3;                 // my_vec: 0, 3, 0, ..., 0
```

Vektoren (mehrdim.)

mehrdimensionale "Massenvariable" 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:** `my_arr.at(1).at(1) = 8 * my_arr.at(0).at(2);`

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

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

Const-Richtlinie
Denke bei jeder Variablen darüber nach, ob sie im Verlauf des Programmes jemals ihren Wert ändern wird oder nicht. Im letzteren Falle verwende das Schlüsselwort **const**, um die Variable zu einer Konstanten zu machen.

Streams

Anmerkung: Ströme dienen dazu, Eingaben aus verschiedenen Quellen (z.B. Konsole, Strings, Dateien) zu holen.

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'
```

std::ifstream — Datentyp für das **Auslesen einer Datei**

Erfordert: `#include<fstream>`

Dient dazu, um Eingaben aus Dateien zu holen.

Objekte des Typs `std::ifstream` können nicht direkt kopiert werden. Deshalb sollte man sie immer via Call-by-Reference an Funktionen übergeben.

```
// Count appearances of 'u' in my_file.txt
std::ifstream reader ("my_file.txt");
// rest of usage is same as for std::cin
char c;
int ctr = 0;
while(reader >> c)
    if(c == 'u')
        ++ctr;
```

std::ostream — Datentyp für **Output-Streams**

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

Beispielsweise `std::cout` hat den Typ `std::ostream`.

Objekte des Typs `std::ostream` können nicht direkt kopiert werden. Deshalb sollte man sie immer via Call-by-Reference an Funktionen übergeben.

```
// POST: wrote the highscore of a given player to out.
void print (std::ostream& out, std::string name, int score) {
    out << "Player: " << name << "  Score: " << score << "\n";
}

int main () {
    print(std::cout, "Pete", 335);
    print(std::cout, "Paula", 410);
    return 0;
}
```

std::ws — **Entfernt Whitespaces** am Anfang eines Input-Streams.

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

```
char c;
std::cin >> std::noskipws; // Do not ignore whitespaces.

// Let's assume the user entered d  a\n\nb
// Output text without whitespaces
std::cin >> c; std::cout << c; // Output: 'd'
std::cin >> std::ws; // Remove: "  "
std::cin >> c; std::cout << c; // Output: 'a'
std::cin >> std::ws; // Remove: "\n\n"
std::cin >> c; std::cout << c; // Output: 'b'
// Output in total: dab
```

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;
```

std::istream — Datentyp für **Input-Streams**

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

Allgemeiner Datentyp, um Input-Ströme zu beschreiben. Man kann ihn sehr gut verwenden, um Funktionen, welche Input-Ströme als Argumente nehmen, unabhängig vom genauen zugrunde liegenden Typ (`std::ifstream`, ...) zu gestalten.

Beispielsweise `std::cin` hat den Typ `std::istream`. Objekte der Typen `std::ifstream` können auch als `std::istream` verwendet werden.

Objekte des Typs `std::istream` können nicht direkt kopiert werden. Deshalb sollte man sie immer via Call-by-Reference an Funktionen übergeben.

```
// POST: Two characters are removed from is. If is contains less
//       characters it is emptied.
void remove_two (std::istream& is) {
    char a;
    is >> a >> a; // remove two chars
}

int main () {
    // Assume that the user enters "Informatics".
    remove_two(std::cin); // istream
    char out;
    while (std::cin >> out)
        std::cout << out; // Output: formatics
    std::ifstream from_file ("my_file.txt");
    remove_two(from_file); // ifstream
    return 0;
}
```

my_stream.peek() — Im Stream nächstes **Zeichen anschauen, ohne es zu entfernen**.

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

Der Rückgabewert ist die int-Repräsentierung des nächsten Zeichens (als `char`) im Stream. **Der Datentyp des Rückgabewerts ist also `int`.**

Diese Funktion ignoriert Whitespaces nie (unabhängig davon, ob der Stream zuerst an `std::noskipws` übergeben wurde oder nicht).

```
std::cin >> std::noskipws; // Do not ignore whitespaces.
char c;

// remove everything before the first 's' (but leave 's' in str)
while (str.peek() != 's')
    std::cin >> c;
// if the user entered "my subst", now we would have "subst"
still in the stream
```

Kinds of loops / Loop controls / Recursion

`for (...) {...}` — for-Schleife

→ optional if just one line of code

Wenn man eine leere Condition als Abbruchbedingung angibt, so wird diese als *wahr* interpretiert.

```
unsigned int n;
std::cin >> n;

// Compute 1 + 2 + 3 + ... + n
unsigned int sum = 0;
for (unsigned int i = 1; i <= n; ++i)
    sum += i;
std::cout << "1 + 2 + ... + n = " << sum << "\n";
```

gets checked at every iteration, if true, loop continues, else, loop stops

exec. at very end of each loop

↳ executed at very beginning of loop

`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";
```

`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.

Die einzelnen Unterscheidungswerte müssen Konstanten sein.

```
std::cout << "Behind which door (1,2,3) is the prize?";
int door_number;
std::cin >> door_number;
switch (door_number) {
    case 1:
    case 3:
        std::cout << "Wrong choice :-(\n";
        break;
    case 2:
        std::cout << "You won the prize!\n";
        break;
    default:
        std::cout << "Error: unknown door number.\n";
}

// User inputs 0 --> Error: unknown door number.
// User inputs 1 --> Wrong choice :-(
// User inputs 2 --> You won the prize!
// User inputs 3 --> Wrong choice :-(
```

`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";
```

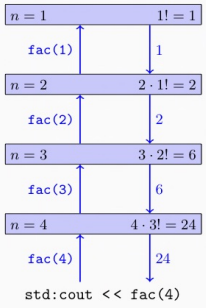
`continue` — zur **nächsten Iteration** springen

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";
```

`Rekursion` — **Selbstaufruf** einer Funktion

Jeder rekursive Funktionsaufruf hat seine eigenen, unabhängigen Variablen und Argumente. Dies kann man sich sehr gut anhand des in der Vorlesung gezeigten Stacks vorstellen (`fac` ist im Beispiel unten definiert):




```
// POST: return value is n!
unsigned int fac (const unsigned int n)
{
    if (n <= 1) return 1;
    return n * fac(n-1); // n > 1
}
```

Structs / Classes + Misc

struct

Container für Datentypen

Wichtige Befehle:

Definition:
```
struct str_name {
    int mem1;
    bool mem2;
    int mem3;
};
```
Objekt erstellen: `str_name obj1;`
mit Startwerten: `str_name obj2 = {3, true, 4};`
aus anderem Objekt: `str_name obj3 = obj2;`
Zugriff auf Member: `obj1.mem1`

Die *Definition* eines Structs hat ein `;` am Schluss.

Nur der Zuweisungsoperator (`=`) wird automatisch erstellt (und kopiert dann die Member einzeln). Die anderen Operatoren (z.B. `==`, `!=`, ...) muss man selbst passend überladen (siehe Eintrag `operator...`).

Bei der Default-Initialisierung eines Objekts des Typs `str_name` werden alle Member einzeln default-initialisiert. Für fundamentale Typen (`int`, `float`, usw.) bedeutet das, dass sie *uninitialisiert* sind, bis man ihnen nachträglich einen Wert zuweist. Das führt zu Problemen, falls man ihren Wert vorher schon ausliest.

```
struct candidate {
    std::string name;      // Name of the participant
    int age;               // Her/his age
};

int main () {
    // Initialization
    candidate mary;                // default-initialisation
    std::cout << mary.age;         // Undefined behavior
    mary.name = "Mary"; mary.age = 43;
    std::cout << mary.age;         // Problem gone: mary.age is 43
    candidate bob = {"Bob", 28}; // using starting values
    candidate fred = bob;        // using other object
    fred.name = "Fred";
    return 0;
}
```

age got default-initialized, which leads to undef-beh. if prompted

```
(a/b)*b + a%b == a   // true
```

no clue why, but it's true

class

Datencontainer mit **Kapselung**

Eine Klasse besteht aus Daten und Funktionen, genannt Member, und erlaubt deren Kapselung via Zugriffkontrolle: Auf Member im privaten Teil (`private`) einer Klasse kann nur durch die Klasse selbst, d.h., deren Member-Funktionen zugegriffen werden.

Zugriff von ausserhalb der Klasse muss über öffentliche (`public`) Member erfolgen. Per default sind die Member einer Klasse privat.

Einziger Unterschied gegenüber `structs`: Member in `structs` sind per default öffentlich (`public`).

Deklarationsreihenfolge von Membern ist irrelevant.

```
class my_class {
public: // public section
    double some_public_member;
private: // private section
    double some_private_member;
};

...
my_class inst;
inst.some_public_member = 1.0;
inst.some_private_member = 0.0; // ERROR: cannot access private
                                //        members directly
```

operator...

Einen **Operator überladen**.

Operator-Überladung wird zum Beispiel verwendet, um Operatoren (`+`, `-`, `*`, etc.) auf eigenen Structs zu definieren.

Mittels dem `operator...` Keyword ist es ebenfalls möglich, den Operator auszuführen. Das sollte man aber vermeiden, da damit der Code unlesbar wird.

```
struct rational {
    int n;
    int d; // INV: d != 0
};

// POST: return value is the sum of a and b
rational operator+ (const rational a, const rational b) {
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}

// POST: return value is the sum of a and b
rational operator+ (const rational a, const int b) {
    rational b_rat;
    b_rat.n = b; b_rat.d = 1; // b_rat is b/1
    return a + b_rat; // Use operator+ for two rationals (above)
}

int main () {
    rational r = {1, 2};
    rational s = {3, 4};
    rational t = r + s; // first overload
    std::cout << t.n << "/" << t.d << "\n"; // Output: 10/8
    rational u = r + 3; // second overload
    std::cout << u.n << "/" << u.d << "\n"; // Output: 7/2
    return 0;
}
```

Standardbibliotheken

```
#include <cmath>
std::pow(2.5,2); // 2.5^2
std::abs(-3.5); // 3.5
std::sqrt(14.0625) // 3.75
#include <algorithm>
std::max(a,b) // returns larger value
!: both have to be same type
```

Constructors

Konstruktor | Datencontainer **Initialisierung**

Konstruktoren sind spezielle Memberfunktionen einer Klasse, die den Namen der Klasse tragen. Sie werden bei der Variablendeklaration aufgerufen.

Sie werden analog zu Funktionen überladen und bei der Variablendeklaration wie eine Funktion aufgerufen. Damit das funktioniert, muss der Konstruktor öffentlich (`public`) sein.

Spezielle Konstruktoren sind der Default-Konstruktor (kein Argument), welcher automatisch erzeugt wird, falls eine Klasse keinen Konstuktor definiert, und der Konversions-Konstruktor (genau ein Argument), welcher die Definition benutzerdefinierter Konversionen ermöglicht.

```
class Insurance {
public:
    Insurance(double v, int r) // general constructor
        : value (v), rate (r)  // initialize data members
        { update_rate(); }
    Insurance()                //default constructor
        : value (0), rate (0)  // initialize data members
        { }
    // other members
private:
    double value;
    double rate;
    void update_rate();
};
...
// General Constructor
Insurance i1 (10000, 10);
// default-Constructor, direct call
Insurance i3; // identical:   Insurance i3 ();
...
-----------------------------------------------------------------
class Complex {
public:
    // Conversion Constructor (float --> Complex)
    Complex(const float i) : real (i), imag (0) { }
private:
    float real;
    float imag;
};
```

shitty var_name choice in the context of complex numbers

Destruktor | Class abbauen

```
// for Customer-struct see example on Copy-Constructor

class Insurance {
public:
    ~Insurance () { delete[] cust; } // free dynamic space
    ...
};
```

Copy-Konstruktor | **Kopier-Initialisierung**

Der Copy-Konstruktor ist der Konstruktor, dessen Argumenttyp `const My_Class&` ist.

```
struct Customer {
    std::string name;
    int duration;
    int amount_insured;
};

class Insurance {
public:
    Insurance (const Insurance& rhs)
        : length (rhs.length), ... // copy remaining data mbrs
        {
            cust = new Customer [length];
            for (int i = 0; i < length; ++i)
                cust[i] = rhs.cust[i];
        }
    ... // other public members
private:
    Customer* cust; // pointer to an array containing customers
    int length;     // length of cust
    ... // other private members
};
```

operator= | **Kopier-Zuweisung**

Eng verwandt mit `operator=` ist der Copy-Konstruktor. Der Unterschied ist, dass der Copy-Konstruktor nur bei der Initialisierung aufgerufen wird, `operator=` hingegen nur *nach* der Initialisierung. z.B.

```
my_class a (5, 6), c (4, 4); // Call a general constructor
my_class b = a;              // Call copy-constructor
c = b;                       // Call operator=
```

`operator=` kann anders als der Copy-Konstruktor implementiert werden müssen. Ein Beispiel sind Klassen, welche Pointer auf dynamisch generierte Objekte als Member haben. Dann muss bei `operator=` meistens zuerst das aktuell vorhandene Objekt gelöscht werden, bevor die Kopie erstellt werden kann. Dies ist beispielsweise beim Stack aus der Vorlesung relevant.

`operator=` gibt im Normalfall eine Referenz auf seinen linken Operanden zurück.

Faustregel: Meistens führt `operator=` zuerst die Aufgaben des Destructors, und dann die Aufgaben des Copy-Konstruktors aus.

```
// for Customer-struct see example on Copy-Constructor

class Insurance {
public:
    Insurance& operator= (const Insurance& rhs) {
        // Cleanup of current customers
        delete[] cust;

        // Copy over the customers from rhs
        cust = new Customer [length];
        for (int i = 0; i < length; ++i)
            cust[i] = rhs.cust[i];
        length = rhs.length;
        ... // copy other data members
        return *this; // return a reference to left operand
    }
    ... // other members
};
```

Memberfunkt. + *this

Memberfunktion | Funktionalität auf Klassen

Memberfunktionen stellen Funktionalität auf einer Klasse bereit. Sie ermöglichen den kontrollierten Zugang zu den privaten Daten und privaten Memberfunktionen. Die *Deklaration* einer Memberfunktion erfolgt immer in der Klassendefinition, die *Definition* der Memberfunktion ist auch extern möglich (ermöglicht vorkompilierte Libraries). Dann muss allerdings die Zugehörigkeit zur Klasse explizit erwähnt werden mittels der `::`-Schreibweise.

Der Aufruf einer Memberfunktion ist `obj.mem_func(arg1, arg2, ..., argN)`. Der Teil `obj.` kann weggelassen werden, falls aus der Class heraus auf einen Member des aufrufenden Objekts (siehe Eintrag `*this`) zugegriffen wird.

```
// Internal Definition vs. External Definition
class Insurance {
public:
    void set_rate_i (const double v) { rate = v; }       // int.
    void set_rate_e (const double v); // Deklaration
    ...
private:
    double rate;
    ...
};

//Definition
void Insurance::set_rate_e (const double v) {rate = v;} // ext.
```

```
// Call from Inside vs. Call from Outside
class Insurance {
public:
    double get_rate () {
        if (!is_up_to_date) update_rate();      // from inside
        return rate;
    }                                  from within class
    double get_cost () {return get_rate() * ...;} // from inside
    ... // e.g. stuff which sets the data members
private:
    bool is_up_to_date;
    double rate;
    double update_rate () { rate = ...; }
};

...
Insurance insurance;
...
std::cout << insurance.get_rate();          // from outside
```

smulere else

const Memberfunktion | Unverändernde Memberfunktion

Das const bezieht sich auf `*this`. Es verspricht, dass durch die Funktionsausführung das implizite Argument nicht im Wert verändert wird.

```
class Insurance {
public:
    double get_value() const {
        return value; // same: return (*this).value;
    }
    ... // e.g. members which set the data members
private:
    double value;
};
```

*this | Zugriff auf **implizites Argument**

Memberfunktionen einer Klasse haben ein implizites Argument, nämlich das aufrufende Objekt. Und `this` ist ein Zeiger darauf. Via `*this` kann man darauf zugreifen.

Bei Zugriffen von innerhalb einer Klasse aus auf Daten-Member oder Member-Funktionen wird das implizite Argument automatisch verwendet. Man muss es dann also nicht unbedingt explizit angeben (siehe Eintrag `Memberfunktion`). Man muss `*this` aber mindestens explizit verwenden, falls z.B. eine Referenz auf das implizite Argument zurückgegeben werden soll.

```
// General example
class Human {
public:
    void set (const int a) { age = a; } // or (*this).age = a;
    void print1 () const { std::cout << (*this).age; }
    void print2 () const { std::cout << age; } // equivalent
private:
    int age;
};
...
Human me;  me.set(175);
me.print1();  // 175
me.print2();  // 175
```

```
// Another example
class Complex {
public:
    // Note: In most applications
    // a reference should be returned.
    Complex& operator+= (const Complex& b) {
        real += b.real;
        imag += b.imag;
        return *this;
    }
    ... // other members
private:
    float real;
    float imag;
};
```

new, delete | **Objekt mit dynamischer Lebensdauer** erstellen.

Mit `new` wird ein Objekt erstellt, indem der nötige Speicherplatz reserviert wird, und dann ein gegebener Konstruktor aufgerufen wird. Bei `delete` wird zuerst ein Destruktor aufgerufen, bevor der Speicherplatz freigegeben wird.

Der Rückgabewert von `new` ist ein *Pointer* auf das neu erstellte Objekt. Wird mit `delete` ein Objekt gelöscht, so sollte man immer *alle* Pointer, die auf das Objekt zeigen, auf `nullptr` setzen.

Jedes `new` braucht ein `delete`. Sonst existieren die erstellten Objekte bis zum Ende des Programms, was je nach Laufdauer eine grosse Speicherverschwendung ist.

```
Class My_Class {
public:
    My_Class (const int i) : y (i) { std::cout << "Hello"; }
    int get_y () { return y; }
private:
    int y;
};

...
My_Class* ptr = new My_Class (3); // outputs Hello
My_Class* ptr2 = ptr; // another pointer to the new object
std::cout << (*ptr).get_y(); // Output: 3
delete ptr;
ptr = nullptr;
ptr2 = nullptr; // has to be done !separately!
...
```

Dyn. Datenstrukturen

new ...[], delete[] | **Ranges mit dynamischer Lebensdauer und Länge** erstellen.

```
int n; std::cin >> n;
int* range = new int[n];
// Read in values to the range
for (int* i = range; i < range + n; ++i) std::cin >> *i;
delete range; // ERROR: must say: delete[]
delete[] range; // This works
```

Pointers / Dynamische Datenstrukturen

Zeiger (generell) | Adresse eines Objekts im Speicher

Wichtige Befehle:

int - pointer

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

(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 **Targets**. Will man also das Target 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 = &a
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.
a_ptr = &a;  // valid
*a_ptr = 9; // a obtains value 9
std::cout << "a == " << a << "\n";       // Output: a == 9
std::cout << "a == " << *a_ptr << "\n"; // Output: a == 9
```

new | Objekt mit dynamischer Lebensdauer erstellen.

Mit `new` wird ein Objekt erstellt, indem der nötige Speicherplatz reserviert wird, und dann ein gegebener Konstruktor aufgerufen wird.

Der Rückgabewert von `new` ist ein *Pointer* auf das neu erstellte Objekt.

```
Class My_Class {
public:
    My_Class (const int i) : y (i) { std::cout << "Hello"; }
    int get_y () { return y; }
private:
    int y;
};

...
My_Class* ptr = new My_Class (3); // outputs Hello
My_Class* ptr2 = ptr; // another pointer to the new object
std::cout << (*ptr).get_y(); // Output: 3
...
```

new ...[] | Ranges mit dynamischer Lebensdauer und Länge erstellen.

```
int n; std::cin >> n;
int* range = new int[n];
// Read in values to the range
for (int* i = range; i < range + n; ++i) std::cin >> *i;
```

const (Zeiger) | Zeiger Konstantheit

Es gibt zwei Arten von Konstantheit:

pointer to (const int)

kein Schreibzugriff auf Target:	`const int* a_ptr = &a;`
kein Schreibzugriff auf Zeiger:	`int* const a_ptr = &a;`

```
int a = 5;
int b = 8;

const int* ptr_1 = &a;
*ptr_1 = 3; // NOT valid (change target)
ptr_1 = &b; // valid (change pointer)

int* const ptr_2 = &a;
*ptr_2 = 3; // valid (change target)
ptr_2 = &b; // NOT valid (change pointer)

const int* const ptr_3 = &a;
*ptr_3 = 3; // NOT valid (change target)
ptr_3 = &b; // NOT valid (change pointer)
```

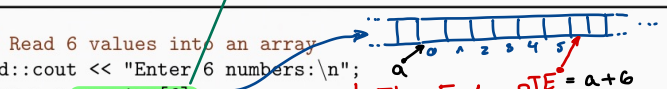
Zeiger | Iterieren

Wichtige Befehle:

Zeiger:	`int* ptr = new int[6];`
temporärer Shift:	`ptr + 3`
	`ptr - 3`
permanenter Shift:	`++ptr`
	`ptr++`
	`--ptr`
	`ptr--`
	`ptr += 3`
	`ptr -= 3`
Distanz bestimmen:	`ptr1 - ptr2`
Position vergleichen:	`ptr1 < ptr2` (Sonst: `<=, >, >=, ==, !=`)

Achtung: Die grünen Shifts erzeugen einen neuen (temporären) Zeiger und verschieben `ptr` **nicht**. Die violetten Shifts verschieben aber `ptr`.

```
// Read 6 values into an array
std::cout << "Enter 6 numbers:\n";
int* a = new int[6];
int* pTE = a+6;
for (int* i = a; i < pTE; ++i)
    std::cin >> *i; // read into array element

// Output:   a[0]+a[3],  a[1]+a[4],  a[2]+a[5]
for (int* i = a; i < a+3; ++i) {
    assert(i+3 < pTE); // Assert that i+3 stays inside.
    std::cout << (*i + *(i+3)) << ",  ";
}
```



...->... | Auf einen Member eines struct/class Objekts zugreifen, auf das ein Pointer gegeben ist.

`ptr->mem` macht das Selbe wie `(*ptr).mem`.

```
struct my_class {
    // POST: "Hi! " is written to std::cout
    void say_hi () const { std::cout << "Hi! "; }
};
...
my_class obj;
my_class* ptr = &obj; // just a pointer to obj
obj.say_hi();         // direct access
ptr->say_hi();        // using ->
(*ptr).say_hi();      // equivalent
```

Pointer / Iterator

Iterator (auf Vektor)

Iterieren über einen Vektor.

Im Folgenden wird nur auf die Unterschiede zum Zeiger (auf Array) eingegangen. Die restliche Bedienung erfolgt gleich.

Erfordert: `#include<vector>`

Wichtige Befehle (gelte `std::vector<int> a (6, 0);`):

Definition: `std::vector<int>::iterator itr = ...;`
Iterator auf `a.at(0)`: `a.begin()`
Past-the-End-Iterator: `a.end()`

Anstelle des `...` in der Definition eines Iterators müssen andere Iteratoren stehen (z.B. `a.begin()`).

```
// Example for vectors.
// To avoid the lengthy lines see entry on typedef.

// Read 6 values into a vector
std::cout << "Enter 6 numbers:\n";
std::vector<int> a (6, 0);
for (std::vector<int>::iterator i = a.begin(); i < a.end(); ++i)
    std::cin >> *i; // read into object of iterator

// Output:   a.at(0)+a.at(3),  a.at(1)+a.at(4),  a.at(2)+a.at(5)
for (std::vector<int>::iterator i = a.begin(); i < a.begin()+3; ++i) {
    assert(i+3 < a.end()); // Assert that i+3 stays inside.
    std::cout << (*i + *(i+3)) << ",  ";
}
```

std::fill(b, p, val)

Wert `val` in einen **Bereich** `[b,p)` **einlesen**

Erfordert: `#include<algorithm>`

```
// Goal: Generate vector: 4 4 4 2 2
std::vector<int> vec (5, 4);              // vec: 4 4 4 4 4
std::fill(vec.begin()+3, vec.end(), 2);   // vec: 4 4 4 2 2
```

std::find(b, p, val)

`val` **suchen** im Bereich `[b,p)`

Erfordert: `#include<algorithm>`

Zurückgegeben wird ein **Iterator** auf das *erste* gefundene Vorkommnis.

Wenn `std::find` nicht fündig wird, gibt es den Past-the-End-Iterator `p` zurück. (Beachte: Past-the-End ist bezüglich Bereich `[b,p)` gemeint.)

```
typedef std::vector<int>::iterator Vit;
std::vector<int> vec (5, 2);
vec.at(3) = -7

// Goal: Find index of -7 in vec: 2 2 2 -7 2
Vit pos_itr = std::find(vec.begin(), vec.end(), -7);
std::cout << (pos_itr - vec.begin()) << "\n"; // Output: 3
```

std::sort(b, e)

Bereich `[b, e)` **sortieren**

Erfordert: `#include<algorithm>`

`std::sort` funktioniert nur, wenn Random-Access Iteratoren für `b` und `e` übergeben werden. Somit funktioniert `std::sort` z.B. für Felder und Vektoren, aber nicht z.B. für Sets.

```
std::vector<int> vec = {8, 1, 0, -7, 7};
std::sort(vec.begin(), vec.end()); // vec:  -7 0 1 7 8
```

const (Iterator)

kein Schreibzugriff auf das Objekt

Vorsicht: Einen `const`-Iterator erzeugt man mittels
`std::vector<int>::const_iterator ...`
und **nicht** mittels
`const std::vector<int>::iterator ...`

Die zweite Version erzeugt einen Iterator, den man nicht herumschieben kann. In dieser Vorlesung gehen wir aber nur auf die Iteratoren näher ein, welche den Schreibzugriff auf *das Objekt* verbieten (erste Variante oben).

Bereichsbasierte for-Schleife

Sequenzielle Iteration mittels eines Iterators über einen `llvec` (const-Iteratormöglich; andere Container möglich):

```
llvec v(3); // v == 0, 0, 0
for (llvec::iterator it = v.begin(); it != v.end(); ++it) {
   std::cout << *it; // 000
}
```

set

Datentyp für **Mengen** (jedes Element kommt nur einmal vor).

Erfordert: `#include<set>`

Wichtige Befehle (Sei `b = some_vec.begin(); e = some_vec.end();`):

Definition: `std::set<int> my_set (b, e);`
(Initialisiert `my_set` mit den Werten im Bereich `[b,e)`.)

Die **Iteratoren** der `set`s funktionieren wie die Iteratoren der Vektoren, **aber:**

Keine: `[], +, -, <, >, <=, >=, +=, -=`
Zum **Verschieben** nur: `++..., ...++, --..., ...--, =`
Zum **Vergleichen** nur: `==, !=`

```
// Determine All Occurring Numbers
std::cout << "Enter 100 numbers:\n";
std::vector<int> nbrs (100);
for (int i = 0; i < 100; ++i)
    std::cin >> nbrs.at(i);

std::set<int> uniques (nbrs.begin(), nbrs.end());

// Output
typedef std::set<int>::iterator Sit;
for (Sit i = uniques.begin(); i != uniques.end(); ++i)
    std::cout << *i << "  ";

// This does not work:
for (int i = 0; i < uniques.end() - uniques.begin(); ++i)
    std::cout << uniques.at(i);
```

using Sit = set::set<int>.

std::min_element(b, p)

Iterator auf Minimum im Bereich `[b,p)`

Erfordert: `#include<algorithm>`

Wenn das Minimum nicht eindeutig ist, so wird ein Iterator auf das erste Vorkommnis zurückgegeben.

```
// Goal: Make sure that all inputs are > 0
std::vector<int> vec (10, 0);
for (int i = 0; i < 10; ++i)
    std::cin >> vec.at(i);

assert( *std::min_element(vec.begin(), vec.end()) > 0 );
    // Note: We have to dereference the (r-value-)iterator.
```