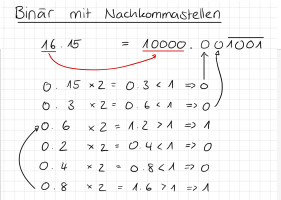
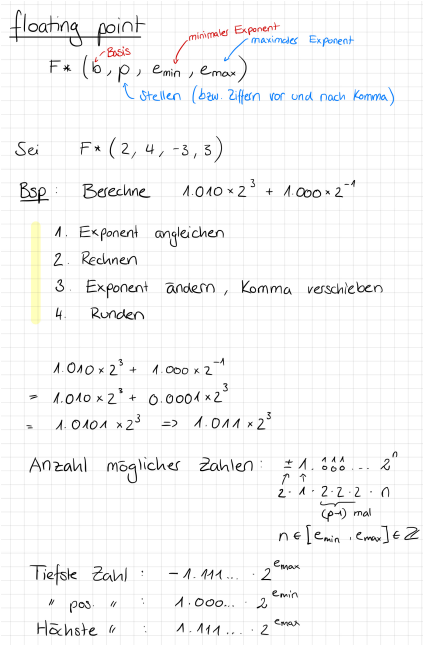
Operatoren/Fliesskommazahlen/Schleifen

- double > float > uns. int > int > char > bool
- Falls Kommazahl in Rechnung: double (NICHT float)
- 1 Byte = 8 Bits und somit mit 4 bytes (2^8)^4 mögliche Ziffern
- uns. int n < 0 —> n=(2^32-n)
- Dezimal in Binär (Beispiel 73):
  - 73%2=1 |36
  - 36%2=0 |18
  - 18%2=0 |9
  - 9%2=1 |4
  - 4%2=0 |2
  - 2%2=0 |1
  - 1%2=1 |0 <— aufhören
- Lösung von unten nach oben lesen: 1001001
- % operator immer gleiches Vorzeichen wie erster Wert

Hexadezimal (16^n):
statt 10,11,12…-> a,b,c…
0x92fe = 14*16^0 + 15*16^1…
Binär: 0b
Hexa: 0x



Bezeichnung	Operatorsymbol	Priorität	Bewertungsreihenfolge
Klammern	() []	14	Von links nach rechts
Komponentenauswahl	. ->	14	Von links nach rechts
Arithmetische Negation	-	13	Von rechts nach links
Logische Negation	!	13	Von rechts nach links
Bitlogische Negation	~	13	Von rechts nach links
Inkrement	++	13	Von rechts nach links
Dekrement	--	13	Von rechts nach links
Arithmetische Operatoren	* / %	12	Von links nach rechts
Arithmetische Operatoren	+ -	11	Von links nach rechts
Shift-Operatoren	<< >>	10	Von links nach rechts
Vergleichsoperatoren	> >= < <=	9	Von links nach rechts
	== !=	8	Von links nach rechts
Bit-Operatoren	&	7	Von links nach rechts
	^	6	Von links nach rechts
	|	5	Von links nach rechts
Logische Operatoren	&&	4	Von links nach rechts
	||	3	Von links nach rechts
Zuweisungsoperatoren	= += -= *= /= %= >>= <<= &= ^= |=	2	Von rechts nach links
Sequenzoperator	,	1	Von rechts nach links



call by value: der Wert wird kopiert
call by reference: ein Alias auf den Wert von einer Variablen wird übergeben

/	**Division**

Präzedenz: 14 und Assoziativität: links

Falls ints oder unsigned ints dividiert werden, so rundet der Operator automatisch zu 0 hin.

```
unsigned int a = 9 / 3;  // Result:  3
unsigned int b = 5 / 3;  // Result:  1
int c = -3 / 2;          // Result: -1
```

%	**Modulo**. Rest der *Ganzzahl*division

Präzedenz: 14 und Assoziativität: links

% gibt es *nur* für int und unsigned int.
Bei negativen Zahlen übernimmt % das Vorzeichen des linken Operanden.

```
int a = 5;
int division = a / 3;    // Result:  1
int rest = a % 3;        // Result:  2
int negative = -5 % -3;  // Result: -2
```

std::numeric_limits<T>::min()	Ermittelt **kleinsten zulässigen Wert** des Datentyps T.

Erfordert: #include<limits>

Sonst gibt es noch:
std::numeric_limits<T>::max()

```
int lower_bdd = std::numeric_limits<int>::min();
std::cout << "Enter a number larger than " << lower_bdd << ": ";
int input;
std::cin >> input; // User knows the smallest valid number.
```

assert	**sofortiges Stoppen** des Programms bei Verletzung einer Bedingung (zu Testzwecken)

Erfordert: #include<cassert>

Wenn das fertige Programm veröffentlicht werden soll, kann man die assert-Befehle bequem deaktivieren.

```
int a;
int b;
std::cin >> a >> b; // read two int values from user
assert(b != 0); // prevent division by 0
std::cout << a / b << "\n";
```

L-Values	R-Values
- Stehen links einer Zuweisung	- Stehen rechts einer Zuweisung
- Haben eine konkrete Adresse im Speicher	- Haben keine Speicheradresse
- Können verändert werden	- Sind nicht veränderbar
- Können auch als R-value verwendet werden	

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
```

my_vec[...]	**Vektor-Zugriff** (Subskript-Operator)

Präzedenz: 17 und Assoziativität: links

Nicht vergessen: Indizes beginnen bei 0 und nicht 1

```
std::vector<int> a = {8, 9, 10, 11};
std::cout << a[0]; // outputs 8
a[3] = 5; // a is 8, 9, 10, 5
```

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
```

```
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 cases 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 :-(
```

Strings/Vektoren

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
```

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::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;
```

```
// 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;
}
```

Class

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;
}
```

```
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;
}
```

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
```

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

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

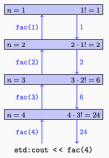
Der Rückgabewert ist die int-Repräsentierung des nächsten Zeichens (als `char`) im Stream. **Der Datentyp des Rückgabewerts ist also** `int`.

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
```

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
}
```

```
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;
};
```

```
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;
}
```

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

Wichtige Befehle:

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

```
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)
```

Pointer/Iteratoren

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)) << ",  ";
}
```

*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 — **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;
```

& — **Adressoperator** (siehe: *Adresse auslesen* unter Zeiger (generell), Summary 8)

* — **Dereferenz-Operator** (siehe: *Zugriff auf Objekt* unter Zeiger (generell))

...->... — **Auf einen Member eines 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
```

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)) << ",  ";
}
```

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).

```
std::vector<int> a (6, -8); // a is: -8 -8 -8 -8 -8 -8

std::vector<int>::const_iterator itr = a.begin() + 3;
*itr = 4;         // NOT valid
itr = a.begin(); // valid (itr now points to a.at(0))
```

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
}
```

Kann alternativ auch wie folgt geschrieben werden:
`for (int i : v) std::cout << i; // 000`
Wird dann zu Iterator-basierter Schleife übersetzt.

Modifizierender Zugriff ist auch möglich:
`for (int& i : v) i += 3;`
`for (int i : v) std::cout << i; // 369`

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 sets 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);
```

Bibliotheken

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

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.
```

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!
...
```

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
};
```

new/delete

`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
```

`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
};
```

`Destruktor`	**Class abbauen**

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

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

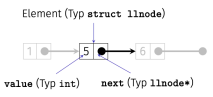
```
#include <iostream>
#include <string>

struct Cat {
  void greet() {
  std::cout << "meow\n";
  }
};

struct Dog {
  void greet() {
  std::cout << "woof\n";
  }
};

int main() {
  Cat* cat = new Cat();
  Dog* dog = new Dog();

  cat->greet();
  dog->greet();

}
```

```
std::vector<int> v = {1, 2, 3};

for (std::vector<int>::iterator it = v.begin();
     it != v.end();
     ++it) {

  *it = -*it;
}

std::cout << v; // -1 -2 -3
```

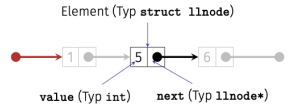
`it` ist ein zu `std::vector<int>` passender Iterator
`it` zeigt initial aufs erste Element
Abbruch falls `it` Ende erreicht hat
`it` elementweise vorwärtssetzen
Aktuelles Element negieren (`x -> -x`)

```
#include <iostream>
#include <vector>
using irow = std::vector<int>;
using matrix = std::vector<irow>;

void output(const matrix& m) {
  for (unsigned int row = 0; row < m.size(); ++row) {
    for (unsigned int col = 0; col < m[0].size(); ++col)
      std::cout << m.at(row).at(col) << " ";
    std::cout << "\n";
  }
}

int main() {
  matrix mat(2, irow(3,0));
  output(mat);
  return 0;
}
```

Node




```
struct llnode {
  int value;
  llnode* next;

  llnode(int v, llnode* n): value(v), next(n) {} // Constructor
};
```




```
class llvec {
  llnode* head;
public: // Public interface identical to avec's
  llvec(unsigned int size);
  unsigned int size() const;
  ...
};
```

```
struct llnode {
  int value;
  llnode* next;
  ...
};
```

```
void llvec::print(std::ostream& sink) const {
  for (llnode* n = this->head;     // Zeiger auf erstes Element
       n != nullptr;               // Abbruch falls Ende erreicht
       n = n->next)                // Zeiger elementweise voranschieben
  {
    sink << n->value << ' ';       // Aktuelles Element ausgeben
  }
}
```

```
int& llvec::operator[](unsigned int i) {
  llnode* n = this->head;          // Zeiger auf erstes Element

  for (; 0 < i; --i)               // Bis zum i-ten voranschreiten
    n = n->next;

  return n->value;                 // i-tes Element zurückgeben
}
```

Konstruktor kann mittels `push_front()` implementiert werden:

```
llvec::llvec(unsigned int size) {
  this->head = nullptr;            // head zeigt zunächst ins Nichts

  for (; 0 < size; --size)         // size mal 0 vorne anfügen
    this->push_front(0);
}
```

Anwendungsbeispiel:

```
llvec v = llvec(3);
std::cout << v; // 0 0 0
```

```
void llvec::push_back(int e) {
  llnode* n = this->head;          // Beim ersten ... und bis zum letzten Element gehen

  for (; n->next != nullptr; n = n->next);

  n->next =                        // Neues Element an bisher letztes anhängen
    new llnode{e, nullptr};
}
unsigned int llvec::size() const {
  unsigned int c = 0;              // Zähler initial 0

  for (llnode* n = this->head;     // Länge der Kette abzählen
       n != nullptr;
       n = n->next)
    ++c;

  return c;                        // Zähler zurückgeben
}
```

```
void llvec::extend(llvec::const_iterator begin, llvec::const_iterator end) {
  if (begin == end) {
    return;
  }
  llvec::const_iterator it = begin;
  if (this->head == nullptr) {
    this->head = new llnode { *it, nullptr };
    ++it;
  }
  llnode *n = this->head;
  while (n->next != nullptr) {
    n = n->next;
  }
  for (; it != end; ++it) {
    n->next = new llnode { *it, nullptr };
    n = n->next;
  }
}
```

```
//Alloziere Speicherblock (Array)
int* array2 = new int[7]; // leer
int* array3 = new int[7]{ 1,2,3,4,5,6,7 }; // mit Werten
```

EBNF

```
train        = "[" ( open | compositions ) "]".
open         = loco cars.
loco         = "*" | "*" loco.
cars         = "-" | "(" cars ")".
compositions = composition { composition }.
composition  = "<" open loco ">".
```

```
bool train(std::istream& is) {
    // train = "[" ( open | compositions ) "]".
    bool valid;
    if (!consume(is, '[')) {
        return false;
    }
    if (lookahead(is) == '<') {
        valid = compositions(is);
    } else {
        valid = open(is);
    }

    // a train formations must end with "]"
    valid = valid && consume(is, ']');

    return valid;
}

bool open(std::istream& is) {
    // open = loco cars .
    return loco(is) && cars(is);
}

bool loco(std::istream& is) {
    // loco = "*" | "*" loco
    bool valid = consume(is, '*');
    if (lookahead(is) == '*') {
        valid = valid && loco(is);
    }
    return valid;
}

bool cars(std::istream& is) {
    // cars = "-" | "(" cars ")"
    bool valid;
    if (lookahead(is) == '(') {
        valid = consume(is, '(');
        valid =  valid && cars(is);
        valid = valid && consume(is, ')');
    }
    else {
        valid = consume(is, '-');
    }
    return valid;
}
```

Verwendung	Zeichen
Definition	=
Aufzählung	,
Endezeichen	;
Alternative	|
Option	[...]
Optionale Wiederholung	{ ... }
Gruppierung	(...)
Anführungszeichen, 1. Variante	" ... "
Anführungszeichen, 2. Variante	' ... '
Kommentar	(* ... *)
Spezielle Sequenz	? ... ?
Ausnahme	-

```
bool compositions(std::istream& is) {
    // compositions = composition | { composition } .
    bool valid = composition(is);
    while (valid && lookahead(is) == '<') {
        valid = valid && composition(is);
    }
    return valid;
}

bool composition(std::istream& is) {
    // composition = "<" open loco ">" .
    return consume(is, '<') && open(is) && loco(is) && consume(is, '>');
}
```

Konstruktoren

Konstruktoren sind spezielle Memberfunktionen. Ihre Aufgabe ist es, sicherzustellen, dass die Objekte der Klasse korrekt **initialisiert** werden. So werden undefined values vermieden. Konstruktoren tragen den Namen der Klasse.

Normalerweise gibt es zwei Arten von Konstruktoren:

1. default constructor (für leere Elemente)
2. Konstruktor für Elemente mit Werten

```
class matrix {
  double a; double b;
  double c; double d;
public:
  matrix() : a(0), b(0), c(0), d(0) {}
  matrix(int A, int B, int C, int D) : a(A), b(B), c(C),
    d(D) {}
```

Beispiel für einen default Konstruktor: `matrix()`
Beispiel für einen Konstruktor mit Werten: `matrix(int A, int B, ...)`

```
  double det() const {
    return this->a * this->d - this->c * this->b;
  }

  matrix inverse() const {
    matrix tmp;
    double mult = 1 / det();
    tmp.a = mult * this->d;
    tmp.b = -mult * this->b;
    tmp.c = -mult * this->c;
    tmp.d = mult * this->a;
    return tmp;
  }
```

Beachte:

- im .h file stehen nur Deklarationen der Klasse
- das .h file muss sowohl ins main file als auch ins .cpp file mittels `#include "classname.h"`eingebunden werden.
- im .cpp file stehen alle Definitionen der Memberfunktionen (und Nicht-Memberfunktionen).
- Alle Memberfunktionen brauchen im .cpp file beim Namen einen Zusatz (damit klar ist, dass sie Memberfunktionen sind). D.h. alle Funktionen heissen im .cpp file `classname::funcname`.

Prioritäten

Bezeichnung	Operatorsymbol	Priorität	Bewertungsreihenfolge
Klammern	() []	14	Von links nach rechts
Komponentenauswahl	. ->	14	Von links nach rechts
Arithmetische Negation	-	13	Von rechts nach links
Logische Negation	!	13	Von rechts nach links
Bitlogische Negation	~	13	Von rechts nach links
Inkrement	++	13	Von rechts nach links
Dekrement	--	13	Von rechts nach links
Arithmetische Operatoren	* / %	12	Von links nach rechts
Arithmetische Operatoren	+ -	11	Von links nach rechts
Shift-Operatoren	<< >>	10	Von links nach rechts
Vergleichsoperatoren	> >= < <=	9	Von links nach rechts
Vergleichsoperatoren	== !=	8	Von links nach rechts
Bit-Operatoren	&	7	Von links nach rechts
Bit-Operatoren	^	6	Von links nach rechts
Bit-Operatoren	|	5	Von links nach rechts
Logische Operatoren	&&	4	Von links nach rechts
Logische Operatoren	||	3	Von links nach rechts
Zuweisungsoperatoren	= += -= *= /= %= >>= <<= &= ^= |=	2	Von rechts nach links
Sequenzoperator	,	1	Von rechts nach links

Beispiele

Tree-Node

```
struct tnode {
  char op; // leaf node: op is '='
           // internal node: op is '+', '-', '*' or '/'
  double val;
  tnode* left;
  tnode* right;

  tnode(char o, double v, tnode* l, tnode* r)
    : op(o), val(v), left(l), right(r) {}
};

int size(const tnode* n){
  if (n){
    return 1 + size(n->left) + size(n->right);
  }
  return 0;
}

// POST: evaluates the subtree with root n
double eval(const tnode* n){
  assert(n);
  if (n->op == '=')
    return n->val;
  double l = 0;
  if (n->left)
    l = eval(n->left);
  double r = eval(n->right);
  switch(n->op){
    case '+': return l+r;
    case '-': return l-r;
    case '*': return l*r;
    case '/': return l/r;
    default: return 0;
  }
}

// POST: a copy of the subtree with root n is made
//       and a pointer to its root node is returned ?\lbl{node}?
tnode* copy (const tnode* n) {
 if (n == nullptr)
   return nullptr;
 return new tnode (n->op, n->val, copy(n->left), copy(n->right));
}

// POST: all nodes in the subtree with root n are deleted
void clear(tnode* n) {
  if(n){
    clear(n->left);
    clear(n->right);
    delete n;
  }
}


private:
 tnode* root;
};

class texpression{
public:
  texpression (double d) : root (new tnode ('=', d, 0, 0)) {}

  ~texpression(){
    clear(root);
  }

  texpression (const texpression& e) : root(copy(e.root)) { }

  texpression& operator=(const texpression& e){
    if (root != e.root){
      texpression cp = e;
      std::swap(cp.root, root);
    }
    return *this;
  }

  texpression& operator-=(const texpression& e) {
    assert (e.root);
    root = new tnode ('-', 0, root, copy(e.root));
    return *this;
  }

  texpression operator-() {
    texpression neg = *this;
    neg.root = new tnode ('-', 0, 0, neg.root);
    return neg;
  }

  texpression& operator+=(const texpression& e) {
    assert (e.root);
    root = new tnode ('+', 0, root, copy(e.root));
    return *this;
  }

  texpression& operator*=(const texpression& e) {
    assert (e.root);
    root = new tnode ('*', 0, root, copy(e.root));
    return *this;
  }

  texpression& operator/=(const texpression& e) {
    assert (e.root);
    root = new tnode ('/', 0, root, copy(e.root));
    return *this;
  }

  double evaluate() const{
    return eval(root);
  }
```

Cell/Box/avec/stack

```
Cell::Cell() {
  value = new int(0);
  std::cerr << "default constructor: value=" << *value << " at " << value << std::endl;
}
Cell::Cell(int val) {
  value = new int(val);
  std::cerr << "overloaded constructor: value=" << *value << " at " << value << std::endl;
}
Cell::~Cell() {
  std::cerr << "destructor: value=" << *value << " at " << value << std::endl;
  delete value;
}
Cell::Cell(const Cell& other) {
  value = new int(*other.value);
  std::cerr << "copy constructor: value=" << *value << " at " << value << std::endl;
}
Cell& Cell::operator= (const Cell& other) {
  *value = *other.value;
  std::cerr << "assignment operator: value=" << *value << " at " << value << std::endl;
  return *this;
}
int Cell::get_value() {
  return *value;
}
void Cell::set_value(int new_value) {
  *value = new_value;
}

Box::Box(Box& other) {
  assert(!other.is_moved());
  ptr = other.ptr;
  other.ptr = nullptr;
  std::cerr << "copy constructor: value=" << *ptr << " at " << ptr << std::endl;
}

Box& Box::operator= (Box& other) {
  assert(!other.is_moved());
  if (this == &other) {
    // Assigning to itself, nothing to do.
    return *this;
  }
  if (!is_moved()) {
    std::cerr << "assignment operator: delete old value=" << *ptr << " at " << ptr << std::endl;
    delete ptr;
    ptr = other.ptr;
  }
  other.ptr = nullptr;
  std::cerr << "assignment operator: new value=" << *ptr << " at " << ptr << std::endl;
  return *this;
}

Box::~Box() {
  if (!is_moved()) {
    std::cerr << "destructor: deallocating value=" << *ptr << " at " << ptr << std::endl;
    delete ptr;
  }
}

Box::Box(int* v) {
  ptr = v;
  std::cerr << "overloaded constructor: value=" << *ptr << " at " << ptr << std::endl;
}

int& Box::value() {
  assert(!is_moved());
  return *ptr;
}

bool Box::is_moved() const {
  return ptr == nullptr;
}

class avec {
  unsigned int count;
  int* elements;

public:
  avec();
  // POST: this contains the same sequence
  // new_element appended at the end.
  void push_back(int new_element);
  void print(std::ostream& sink) const;
};

  avec::avec(): count(0) {
    this->elements = new int[0];
  }

  // PRE: source_begin points to the first element to be copied.
  // PRE: source_ends points to the element after the last element to be copied.
  // PRE: destination_begin points to the first element of the memory block into
  //      which the elements from the source should be copied.
  // PRE: The target memory location has at least as many elements as there are
  //      elements between source_begin and source_end.
  void copy_range(
    const int* const source_begin,
    const int* const source_end,
    int* const destination_begin
  ) {
    int* dst = destination_begin;
    for (const int* src = source_begin; src != source_end; ++src) {
      *dst = *src;
      ++dst;
    }
  }

  void avec::push_back(int new_element) {
    int* const new_elements = new int[this->count + 1];
    copy_range(this->elements, this->elements + this->count, new_elements);
    new_elements[this->count] = new_element;
    this->count++;
    this->elements = new_elements;
    // Note: The complete solution should also deallocate the old memory
    // block. Deallocation will be covered later.
  }

  void avec::print(std::ostream& sink) const {
    const int* const begin = this->elements;
    const int* const end = this->elements + this->count;
    for (const int* p = begin; p != end; ++p) {
      sink << *p << ' ';
    }
    sink << '\n';
  }
```

```
class stack{
public:
  stack() : topn(nullptr) {}

  // POST: *this is initialized with a copy of s
  stack (const stack& s) : topn (nullptr) {
    if (s.topn == nullptr)
      return;
    topn = new llnode(s.topn->value, nullptr);
    llnode* prev = topn;
    for(llnode* n = s.topn->next; n != nullptr; n = n->next){
      llnode* copy = new llnode(n->value, nullptr);
      prev->next = copy;
      prev = copy;
    }
  }

  // POST: *this (left operand) becomes a
  //             copy of s (right operand)
  stack& operator= (const stack& s){
    if (topn != s.topn){ // no self-assignment
      stack copy = s; // Copy Construction
      std::swap(topn, copy.topn);
      // now *this has the copied data
      // and copy has the garbage and
    } // copy is cleaned up automatically
    return *this;
  }

  // post: deconstruct (clean-up) stack
  ~stack(){
    while (topn != nullptr){
      llnode* t = topn;
      topn = t->next;
      delete t;
    }
  }

  // post: Push an element onto the stack
  void push(int value){
    topn = new llnode(value, topn);
  }
  // pre: non-empty stack
  // post: Delete top most element from the stack
  void pop(){
    assert(!empty());
    llnode* p = topn;
    topn = topn->next;
    delete p;
  };

  // pre: non-empty stack
  // post: return value of top most element
  int top() const{
    assert(!empty());
    return topn->value;
  }
  // post: return if stack is empty
  bool empty() const{
    return topn == nullptr;
  }
  // post: print out the stack
  void print(std::ostream& out) const {
    for(const llnode* p = topn; p != nullptr; p = p->next)
      out << p->value << " ";
  }
private:
  llnode* topn;
};
```

.h file

```
class matrix {
  double a;
  double b;
  double c;
  double d;

public:
  matrix();
  matrix(int A, int B,
    int C, int D);

  matrix sum(const
    matrix& m2) const;

  double det() const;

  matrix inverse()
    const;

  void print() const;
};
```

```
#include <iostream>
#include "class_kurz_split.h"

matrix::matrix() : a(0), b(0), c(0), d(0) {}
matrix::matrix(int A, int B, int C, int D) :
    a(A), b(B), c(C), d(D) {}

matrix matrix::sum(const matrix& m2) const {
  matrix tmp;
  tmp.a = this->a + m2.a;
  // [...]
  return tmp;
}
double matrix::det() const {
  return this->a * this->d - this->c * this->b;
}
matrix matrix::inverse() const {
  matrix tmp;
  double mult = 1 / det();
  tmp.a = mult * this->d;
  tmp.b = -mult * this->b;
  // [...]
  return tmp;
}
void matrix::print() const {
  std::cout << this->a << /*[...]*/;
}
```

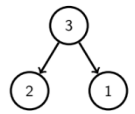
llvec

```
class llvec {
  // llvec uses a linked-list to represent its vector elements.
  // llnodes are private to llvec, and only hold data but don't provide any
  // functionality. We therefore implement them as structs, not classes
  // (although there is no operational difference between structs and classes,
  // except the default member visibility).
  struct llnode {
    int value; // Element value
    llnode* next; // Next element
  };

  llnode* head; // First linked-list node
  llvec::llvec(unsigned int size) {
    this->head = nullptr;

    for (; 0 < size; --size)
      this->push_front(0);
  }

  unsigned int llvec::size() const {
    unsigned int s = 0;

    for (llnode* n = this->head; n != nullptr; n = n->next)
      ++s;

    return s;
  }

  int& llvec::getElement(unsigned int i) const {
    llnode* n = this->head;

    for (; 0 < i; --i)
      n = n->next;

    return n->value;
  }

  int& llvec::operator[](unsigned int i) {
    return this->getElement(i);
  }

  const int& llvec::operator[](unsigned int i) const {
    return this->getElement(i);
  }

  int& llvec::at(unsigned int i) {
    assert(i < this->size());

    return this->getElement(i);
  }

  const int& llvec::at(unsigned int i) const {
    assert(i < this->size());

    return this->getElement(i);
  }

  void llvec::print(std::ostream& sink) const {
    for (llnode* n = this->head; n != nullptr; n = n->next)
      sink << n->value << ' ';

    sink << '\n';
  }
  void llvec::push_front(int e) {
    this->head = new llnode(e, this->head);
  }

  void llvec::push_back(int e) {
    llnode* n = this->head;

    for (; n->next != nullptr; n = n->next);

    n->next = new llnode(e, nullptr);
  }

  llvec::iterator llvec::begin() {
    return llvec::iterator(this->head);
  }

  llvec::iterator llvec::end() {
    return llvec::iterator(nullptr);
  }

  llvec::const_iterator llvec::cbegin() const {
    return llvec::const_iterator(this->head);
  }

  llvec::const_iterator llvec::cend() const {
    return llvec::const_iterator(nullptr);
  }

  // ----- llvec::iterator -----

  llvec::iterator::iterator(llnode* n): node(n) {}

  llvec::iterator& llvec::iterator::operator++() {
    assert(this->node != nullptr);

    this->node = this->node->next;

    return *this;
  }

  int& llvec::iterator::operator*() const {
    return this->node->value;
  }

  bool llvec::iterator::operator!=(const llvec::iterator& other) const {
    return this->node != other.node;
  }


  // ----- llvec::const_iterator -----

  llvec::const_iterator::const_iterator(llnode const* n): node(n) {}

  llvec::const_iterator& llvec::const_iterator::operator++() {
    assert(this->node != nullptr);

    this->node = this->node->next;

    return *this;
  }

  const int& llvec::const_iterator::operator*() const {
    return this->node->value;
  }

  bool llvec::const_iterator::operator!=(const llvec::const_iterator& other) const {
    return this->node != other.node;
  }

  // ----- Overloaded external operators -----

  std::ostream& operator<<(std::ostream& sink, const llvec& vec) {
    vec.print(sink);

    return sink;
  }
```

alte Aufgaben

```
// PRE:   root points to a (potentially empty) binary tree, with all keys > 0.
// POST: Returns -1 if the tree violates the desired property, and any
//       number != -1 otherwise.
int validate(Node const * const root) {
  // empty tree
  if (root == nullptr) return 0;
  // leaf node
  if (root->left == nullptr && root->right == nullptr){
    return root->key;
  }
  // calculate children
  int left = validate(root->left);
  int right = validate(root->right);
  // check condition
  if (left != -1 && right != -1 && root->key == left + right){
    return root->key;
  }
  // default
  return -1;
}
```

T1




```
// post: add a new node with value to the sorted linked list
//       several nodes with the same value are possible
void sorted_list::add(int value){
  // this creates a new node in dynamic memory, wrapped in a shared pointer
  // analogously to new node(value) for normal pointers
  node_ptr newNode = std::make_shared<node>(value);

  node_ptr prev = nullptr;
  node_ptr n = first;
  while (n != nullptr && n->value < value){
    prev = n;
    n = n->next;
  }

  if (prev == nullptr){
    first = newNode;
    newNode->next = n;
  } else {
    prev->next = newNode;
    newNode->next = n;
  }
}

// post: remove one node with value, if any
//       if there is no such node, return false
//       otherwise return true
bool sorted_list::remove(int value){
  node_ptr prev = nullptr;
  node_ptr n = first;
  while (n != nullptr && n->value != value){
    prev = n;
    n = n->next;
  }

  if(n == nullptr) {
    // didn't find it
    return false;
  }

  if (prev == nullptr){
    first = n->next;
  }
  else {
    prev->next = n->next;
  }

  return true;

}
// return the union of two sets l and r
Set operator+(Set l, const Set& r){
  return l+=r; // replace by correct implmementation
}

// intersect this set with r and return it
Set& Set::operator+=(const Set& r){
  for (unsigned int i = 0; i<elements.size(); ++i){
    elements[i] = elements[i] && r.elements[i];
  }
  return *this;
}

// return the intersection of two sets
Set operator*(Set l, const Set& r){
  return l *= r;// replace by correct implmementation
}

// return if two sets are unequal
bool operator!=(const Set& a, const Set& b){
  return !(a==b); // replace by correct implmementation
}

// return if set a is contained in b
bool operator<=(const Set& a, const Set& b){
  return a<b == a; // replace by correct implmementation
}

// return if set a is a superset of b
bool operator>=(const Set& a, const Set& b){
  return a>b == b; // replace by correct implmementation
}

// return if a is a strict subset of b
bool operator<(const Set& a, const Set& b){
  return (a<b == a) & (a!=b); // replace by correct implmementation
}

// POST: Find and return the maximum leaf node value in a (sub)tree starting
//       at root. If root is the empty tree, return 0.
unsigned int findMax(const Node* root) {
  unsigned int max = 0;

  // If the tree is empty, return 0
  if (root == nullptr) {
    return 0;
  }

  // If the current node is a leaf, return its value
  if (root->isLeaf()) {
    return root->value;
  }

  // Otherwise, the current node is not a leaf, and we search for the max
  // in its children
  for (Node* child : root->children) {
    unsigned int childMax = findMax(child);

    if (childMax > max) {
      max = childMax;
    }
  }

  return max;
}
```

EBNF

```
// digit = '0' | '1' | ... | '9'.
bool isDigit(char ch) {
  return ch >= '0' && ch <= '9';
}

// PRE:  valid input stream with a digit as next consumable character
// POST: returns the number consumed from the stream
// number = digit ... . // depends on number type
bool number(std::istream& input) {
  char ch = lookahead(input);
  if (!isDigit(ch))
    return false;
  number_type value;
  input >> value; // use the builtin
  return !input.fail();
}

// POST: returns if from stream input a
//       valid expression could be consumed
// expression =  term { "+" term | "-" term }
bool expression (std::istream& input)
{
  if (!term(input)) return false; // term
  while (consume(input,'+') || consume (input, '-')) // {"+" | "-"
    if (!term(input)) return false; // term}
  return true;
}

// POST: returns if from stream input a
//       valid term could be consumed according to
// term = factor { "*" factor | "/" factor }
bool term (std::istream& input)
{
  if (!factor(input)) return false; // factor
  while (consume(input,'*') || consume (input, '/')) // {"*"|"/"
    if (!factor(input)) return false; // factor }
  return true;
}

// POST: returns if from stream input a
//       valid factor could be consumed according to
// factor = "(" expression ")" | "-" factor | number
bool factor (std::istream& input)
{
  if (consume (input, '(')) { // "("
    if (!expression(input)) return false; // factor
    if (!consume(input, ')')) return false; // ")"
  } else if (consume (input, '-')){ // "-"
    if (!factor(input)) return false; // factor
  } else { // number
    return number(input);
  }
  return true;
}
```

```
// PRE:  valid input stream input
// POST: returns true if further input is available
//       otherwise false
bool input_available(std::istream& input);

// PRE:  valid input stream input
// POST: the next character at the stream is returned (but not consumed)
//       if no input is available, 0 is returned
char peek(std::istream& input);

// POST: leading whitespace characters are extracted
//       from input, and the first non-whitespace character is returned (but not consumed)
//       if an error or end of stream occurs, 0 is returned
char lookahead(std::istream& input);

// PRE:  Valid input stream input, expected > 0
// POST: If ch matches the next lookahead then it is consumed and true is returned
//       otherwise no character is consumed and false is returned
bool consume(std::istream& input, char expected);
```

```
// PRE:  Valid input stream is.
// POST: Returns true if stream contains term and extracts it,
//       otherwise false. term = "A" { "a" } | "a" { "a" }
bool term(std::istream &is) {
  if (consume(is, 'A')) {
    while (consume(is, 'a')) {};
    return true;
  }
  else if (consume(is, 'a')) {
    while (consume(is, 'a')) {};
    return true;
  }
  else {
    return false;
  }
}


// PRE:  Valid input stream is.
// POST: Returns true if stream contains a "seq" and extracts it,
//       otherwise false. seq = term [ "_" seq ]
bool seq(std::istream &is) {
  if (term(is)) {
    return (peek(is) == 0 || (consume(is, '_') && seq(is)));
  }
  return false;

  // Notice how we use peek() rather than lookahead() in this exercise.
  // This is because in our grammar we do *not* want to ignore whitespaces.
  // This is different than the Calculator exercise you have seen in class,
  // where whitespaces were not considered relevant for the input.
}
```

Rekursion

```
// POST: Prints instructions how to transfer n discts from src to dst.
void move(int n, const std::string &src, const std::string &aux, const std::string &dst) {
  if (n == 1) {
    // base case ('move' the disc)
    std :: cout << src << " --> " << dst << std::endl;
  } else {
    // recursive case
    move(n-1, src, dst, aux);
    move(1, src, aux, dst );
    move(n-1, aux, src, dst);
  }
}
```

```
 // function to print leaf
 // nodes from left to right
 void printLeafNodes(Node *root)
 {
     // if node is null, return
     if (!root)
         return;

     // if node is leaf node, print its data
     if (!root->left && !root->right)
     {
         cout << root->data << " ";
         return;
     }

     // if left child exists, check for leaf
     // recursively
     if (root->left)
        printLeafNodes(root->left);

     // if right child exists, check for leaf
     // recursively
     if (root->right)
        printLeafNodes(root->right);
 }
```

```
// C++ program to generate all binary strings with
// equal sums in left and right halves.
#include <bits/stdc++.h>
using namespace std;

/* Default values are used only in initial call.
   n is number of bits remaining to be filled
   di is current difference between sums of
      left and right halves.
   left and right are current half substrings. */
void equal(int n, string left="", string right="",
                                        int di=0)
{
    // TWO BASE CASES
    // If there are no more characters to add (n is 0)
    if (n == 0)
    {
        // If difference between counts of 1s and
        // 0s is 0 (di is 0)
        if (di == 0)
            cout << left + right << " ";
        return;
    }

    /* If 1 remains than string length was odd */
    if (n == 1)
    {
        // If difference is 0, we can put remaining
        // bit in middle.
        if (di == 0)
        {
            cout << left + "0" + right << " ";
            cout << left + "1" + right << " ";
        }
        return;
    }

    /* If difference is more than what can be
       be covered with remaining n digits
       (Note that lengths of left and right
        must be same) */


    if ((2 * abs(di) <= n))
    {

        /* add 0 to end in both left and right
        half. Sum in both half will not
                change*/
        equal(n-2, left+"0", right+"0", di);

        /* add 0 to end in both left and right
        half* subtract 1 from di as right
        sum is increased by 1*/
        equal(n-2, left+"0", right+"1", di-1);

       /* add 1  in end in left half and 0 to the
       right half. Add 1 to di as left sum is
       increased by 1*/
       equal(n-2, left+"1", right+"0", di+1);

       /* add 1 in end to both left and right
          half the sum will not change*/
       equal(n-2, left+"1", right+"1", di);
    }
```

Anderes

```
void Queue::enqueue(int value) {
  is_valid();
  Node* node = new Node(value, nullptr);
  if (first == nullptr) {
    first = node;
    last = node;
  } else {
    last->next = node;
    last = node;
  }
}

int Queue::dequeue() {
  is_valid();
  assert(!is_empty());

  Node* node = first;
  first = first->next;
  if (first == nullptr) {
    last = nullptr;
  }

  int value = node->value;
  // A proper solution should also free the memory used by the first node.
  // In the upcoming week, we will learn the operator delete that can be
  // used for this purpose:
  // delete node;

  return value;
}

bool Queue::is_empty() const {
  is_valid();
  return first == nullptr;
}
```

Dezimal	Hexadezimal	Oktal	Binär/Dual
0	00	0	00000000
1	01	1	00000001
2	02	2	00000010
3	03	3	00000011
4	04	4	00000100
5	05	5	00000101
6	06	6	00000110
7	07	7	00000111
8	08	10	00001000
9	09	11	00001001
10	0A	12	00001010
11	0B	13	00001011
12	0C	14	00001100
13	0D	15	00001101
14	0E	16	00001110
15	0F	17	00001111
16	10	20	00010000
17	11	21	00010001
18	12	22	00010010
19	13	23	00010011
20	14	24	00010100
21	15	25	00010101
22	16	26	00010110
23	17	27	00010111
24	18	30	00011000
25	19	31	00011001
26	1A	32	00011010
27	1B	33	00011011
28	1C	34	00011100
29	1D	35	00011101
30	1E	36	00011110
31	1F	37	00011111
32	20	40	00100000
33	21	41	00100001
34	22	42	00100010
35	23	43	00100011
36	24	44	00100100
37	25	45	00100101
38	26	46	00100110
39	27	47	00100111
40	28	50	00101000
41	29	51	00101001
42	2A	52	00101010
43	2B	53	00101011
44	2C	54	00101100
45	2D	55	00101101
46	2E	56	00101110
47	2F	57	00101111
48	30	60	00110000
49	31	61	00110001
50	32	62	00110010
51	33	63	00110011
101	65	145	01100101
102	66	146	01100110
103	67	147	01100111
104	68	150	01101000
105	69	151	01101001
106	6A	152	01101010
107	6B	153	01101011
108	6C	154	01101100
109	6D	155	01101101
110	6E	156	01101110
111	6F	157	01101111
112	70	160	01110000
113	71	161	01110001
114	72	162	01110010
115	73	163	01110011
116	74	164	01110100
117	75	165	01110101
118	76	166	01110110
119	77	167	01110111
120	78	170	01111000
121	79	171	01111001
122	7A	172	01111010
123	7B	173	01111011
124	7C	174	01111100
125	7D	175	01111101
126	7E	176	01111110
127	7F	177	01111111
128	80	200	10000000
129	81	201	10000001
130	82	202	10000010
131	83	203	10000011
132	84	204	10000100
133	85	205	10000101
134	86	206	10000110
135	87	207	10000111
136	88	210	10001000
137	89	211	10001001
138	8A	212	10001010
139	8B	213	10001011
140	8C	214	10001100
141	8D	215	10001101
142	8E	216	10001110
143	8F	217	10001111
144	90	220	10010000
145	91	221	10010001
146	92	222	10010010
147	93	223	10010011
148	94	224	10010100
149	95	225	10010101
150	96	226	10010110
151	97	227	10010111
152	98	230	10011000
153	99	231	10011001
154	9A	232	10011010
155	9B	233	10011011
156	9C	234	10011100
157	9D	235	10011101
158	9E	236	10011110
159	9F	237	10011111
160	A0	240	10100000
161	A1	241	10100001
162	A2	242	10100010
163	A3	243	10100011
164	A4	244	10100100
165	A5	245	10100101
166	A6	246	10100110
167	A7	247	10100111
168	A8	250	10101000
169	A9	251	10101001
170	AA	252	10101010
171	AB	253	10101011
172	AC	254	10101100
173	AD	255	10101101
174	AE	256	10101110
175	AF	257	10101111
176	B0	260	10110000
177	B1	261	10110001
178	B2	262	10110010
179	B3	263	10110011
180	B4	264	10110100
181	B5	265	10110101
182	B6	266	10110110
183	B7	267	10110111
184	B8	270	10111000
185	B9	271	10111001
186	BA	272	10111010
187	BB	273	10111011
188	BC	274	10111100
189	BD	275	10111101
190	BE	276	10111110
191	BF	277	10111111
192	C0	300	11000000
193	C1	301	11000001
194	C2	302	11000010
195	C3	303	11000011
196	C4	304	11000100
197	C5	305	11000101
198	C6	306	11000110
199	C7	307	11000111
200	C8	310	11001000
201	C9	311	11001001
202	CA	312	11001010
203	CB	313	11001011
204	CC	314	11001100
205	CD	315	11001101
206	CE	316	11001110
207	CF	317	11001111
208	D0	320	11010000
209	D1	321	11010001
210	D2	322	11010010
211	D3	323	11010011
212	D4	324	11010100
213	D5	325	11010101
214	D6	326	11010110
215	D7	327	11010111
216	D8	330	11011000
217	D9	331	11011001
218	DA	332	11011010
219	DB	333	11011011
220	DC	334	11011100
221	DD	335	11011101
222	DE	336	11011110
223	DF	337	11011111
224	E0	340	11100000
225	E1	341	11100001
226	E2	342	11100010
227	E3	343	11100011
228	E4	344	11100100
229	E5	345	11100101
230	E6	346	11100110
231	E7	347	11100111
232	E8	350	11101000
233	E9	351	11101001
234	EA	352	11101010
235	EB	353	11101011
236	EC	354	11101100
237	ED	355	11101101
238	EE	356	11101110
239	EF	357	11101111
240	F0	360	11110000
241	F1	361	11110001
242	F2	362	11110010
243	F3	363	11110011
244	F4	364	11110100
245	F5	365	11110101
246	F6	366	11110110
247	F7	367	11110111
248	F8	370	11111000
249	F9	371	11111001
250	FA	372	11111010
251	FB	373	11111011
252	FC	374	11111100
253	FD	375	11111101
254	FE	376	11111110
255	FF	377	11111111

Scan-code	ASCII hex	dez	Zeichen
	00	0	NUL ^@
	01	1	SOH ^A
	02	2	STX ^B
	03	3	ETX ^C
	04	4	EOT ^D
	05	5	ENQ ^E
	06	6	ACK ^F
	07	7	BEL ^G
0E	08	8	BS ^H
0F	09	9	TAB ^I
	0A	10	LF ^J
	0B	11	VT ^K
	0C	12	FF ^L
1C	0D	13	CR ^M
	0E	14	SO ^N
	0F	15	SI ^O
	10	16	DLE ^P
	11	17	DC1 ^Q
	12	18	DC2 ^R
	13	19	DC3 ^S
	14	20	DC4 ^T
	15	21	NAK ^U
	16	22	SYN ^V
	17	23	ETB ^W
	18	24	CAN ^X
	19	25	EM ^Y
	1A	26	SUB ^Z
01	1B	27	Esc ^[
	1C	28	FS ^\
	1D	29	GS ^]
	1E	30	RS ^^
	1F	31	US ^_

Scan-code	ASCII hex	dez	Zch.
	20	32	SP
02	21	33	!
03	22	34	"
29	23	35	#
05	24	36	$
06	25	37	%
08	26	38	&
0D	27	39	'
09	28	40	(
0A	29	41	)
1B	2A	42	*
1B	2B	43	+
33	2C	44	,
35	2D	45	-
34	2E	46	.
08	2F	47	/
0B	30	48	0
02	31	49	1
03	32	50	2
04	33	51	3
05	34	52	4
06	35	53	5
07	36	54	6
08	37	55	7
09	38	56	8
0A	39	57	9
34	3A	58	:
33	3B	59	;
2B	3C	60	<
0B	3D	61	=
2B	3E	62	>
0C	3F	63	?

Scan-code	ASCII hex	dez	Zch.
	40	64	@
1E	41	65	A
30	42	66	B
2E	43	67	C
20	44	68	D
12	45	69	E
21	46	70	F
22	47	71	G
23	48	72	H
17	49	73	I
24	4A	74	J
25	4B	75	K
26	4C	76	L
32	4D	77	M
31	4E	78	N
18	4F	79	O
19	50	80	P
10	51	81	Q
13	52	82	R
1F	53	83	S
14	54	84	T
16	55	85	U
2F	56	86	V
11	57	87	W
2D	58	88	X
15	59	89	Y
2C	5A	90	Z
	5B	91	[
	5C	92	\
	5D	93	]
29	5E	94	^
35	5F	95	_

Scan-code	ASCII hex	dez	Zch.
0D	60	96	`
1E	61	97	a
30	62	98	b
2E	63	99	c
20	64	100	d
12	65	101	e
21	66	102	f
22	67	103	g
23	68	104	h
17	69	105	i
24	6A	106	j
25	6B	107	k
26	6C	108	l
32	6D	109	m
31	6E	110	n
18	6F	111	o
19	70	112	p
10	71	113	q
13	72	114	r
1F	73	115	s
14	74	116	t
16	75	117	u
2F	76	118	v
11	77	119	w
2D	78	120	x
15	79	121	y
	7A	122	z
	7B	123	{
	7C	124	|
	7D	125	}
	7E	126	~
53	7F	127	DEL

Allgemeines:

Präzedenzen (Auswahl)

Präz.	Operator
1	`::` (Bereichsauflösung)
2	`a++, a--, (), [], ., ->`
3	`++a, --a, -, !, *, &, new, delete`
4	`.>, *>` (Pointer to Member)
5	`*, /, %`
6	`+, -`
8	`>, >=, <, <=`
9	`==, !=`
13	`&&` (logisches UND)
14	`||` (logisches ODER)
15	`=, +=, -=, *=, /=, %=`

Nützliche Funktionen

- `#include <cassert>`

```
#define NDEBUG zum ausschalten
```

  Assertions brechen das Programm ab wenn das Statement in `assert(bool)` false bzw. =0 ist.
- `std::to_string(T)` gibt den zugehörigen String zu den meisten Typen zurück.
- `push_back(T)` gibt dem vector einen neuen Platz für das Element T
- `bool std::find(it begin, it end, T);` true, wenn T im Container
- **Aus istream Inhalt entfernen**: `stream >> c;`

Typ- Alias

```
using Name = Typ;
using int_vec = std::vector<int>;
using Cvit = int_vec::const_iterator;
→ Cvit iterator = vector.begin();
```

Variablen:

L-Wert und R-Wert

R-Werte sind z.B. Ergebnisse einer Operation, während L-Werten ein R-Wert zugewiesen werden kann.
Achtung «schlechte Ausdrücke» : `(a+b)*(a++)` hängt bei der Auswertungsreihenfolge vom Compiler ab ob zuerst a++ oder a+b berechnet wird!

Arithmetische Operatoren und Zuweisung:

Arithmetische Operatoren (`*, /, %, +, -`) sind linksassoziativ: bei gleicher Präzedenz erfolgt Auswertung von links nach rechts. → C++ rechnet selbst «Punkt vor Strich»!
Alle Operatoren: **[R-Wert x] R-Wert → R-Wert**

Verkürzte Operatoren/ In- und Decrement Operator

`x += 1;` ↔ `x = x + 1;` gilt für: `+-*/%`
`c++;` & `++c;` ↔ `c = c + 1;` gilt für: `++ --`
Achtung: `c++` gibt den alten Wert (als R- Wert) `++c` den neuen Wert (als L-Wert) zurück!

Binärsystem

Ganze Zahlen

Binäre Darstellung ist in der Basis 2 (aus {0, 1})

$$b_n b_{(n-1)} \dots b_1 b_0 \leftrightarrow b_n \cdot 2^n + \dots + b_1 \cdot 2 + b_0$$

Wertebereich bei int, Überlauf

$$-2^{(B-1)}, -2^{(B-1)} + 1, \dots, -1, 0, 1, \dots, 2^{(B-1)} - 2, 2^{(B-1)} - 1$$

Unsigned int: $0, 1, \dots, 2^{(B)} - 1$
Arithmetische Operationen können ohne Fehlermeldung aus dem Wertebereich hinaus führen!
Tipp: $int - a = uint\ 2^B - a$ →aus Modulorechnung

Fliesskommazahlen Double / float

Kein Modulo-Operator (`%, %=`)
Literale: 1.0, 1.2e-7 → double; 1.0f, 1.2e-7f →float
Fliesskommazahlensysteme
Definiert durch $F * (\beta, p, e_{min}, e_{max})$
β ≥ 2, die Basis; p ≥ 1, die Präzision (Stellenzahl); emin, der kleinste Exponent; emax, der grösste Exponent
`float`: $F * (2, 24, -126, 127)$ → 32 Bit
`double`: $F * (2, 52, -1022, 1023)$ → 64 Bit
Jeweils im Exponenten 2 Spezialwerte: 0, ∞

Hexadezimalsystem

Binäre Darstellung ist in der Basis 16 ({0, …, 9, a, …, f})

$$h_n h_{(n-1)} \dots h_1 h_0 \leftrightarrow h_n \cdot 16^n + \dots + h_1 \cdot 16 + h_0$$

→ ein hex-Nibble (h) entspricht genau 4 Bits

Konversion zu binären Kommazahlen für Zahlen > 2

$$i = 0, \quad b_i = \begin{cases} 1 \ if \ x_i \geq 1 \\ 0 \quad else \end{cases} \quad x_{i+1} = 2(x_i - b_i)$$

Solange $b_i \neq 0$ oder keine Wiederholung. Danach:

$$z = b_0.b_1 b_2 \dots$$

Datentypen

Boolean und Operatoren

`&&`	Logisches UND `AND`
`||`	Logisches ODER `OR`
`!`	Logisches NICHT (invertieren) `NOT`

Daraus können alle anderen boolschen Funktionen erzeugt werden. Präzedenzen beachten!
Charakteristischer Vektor z.B. bei XOR: f_0110

Conversion (Zahlen und Wahrheitswerte)

bool → int: `true → 1, false → 0`
int → bool: `≠0→ true, 0 → false`

Char und ASCII

Char ist ein 8-Bit langer Wert, repräsentiert druckbare Zeichen und Steuerzeichen (z.B. '\n') und kann zu int bzw. unsigned int konvertiert: `char ch = 'a';`
char haben immer ein Apostroph (‘), Strings zwei (“).
ASCII definert die Konversationsregeln von int bzw. unsigned int zu char. UTF-8 erweitert dieses System auf verschiedenste Schriftsysteme.
Tipp: in Grossbuchstaben umwandelt:

```
gross_char = klein_char + ('a' - 'A');
```

Buchstaben verschieben:

```
char c = 'a'; c = c + 4; //e
```

Funktionen:

Kontrollanweisung

Allgemein: falls nur ein `statement`, kann man `{…}` weglassen

If-/ Else- Anweisung, switch

```
if ( condition ){
    statement
}else if (condition){
    statement
} else {
    statement
}
```


Switch

```
switch (L-Value) {
   case R-Value: statements;
                 break;
   default: statements;
}
```

Soviele cases wie man will, falls break fehlt wird nächster case ausgeführt. Case wird ausgeführt, wenn der jeweilige R-Value dem L-Value entspricht.

Loops / Iterationen

```
for (init statement; cond.; expression){
     statement
}
```

init statement wird ausgeführt → von da an solange cond. == true →statement wird ausgeführt, expression wird ausgeführt, condition überprüft, …

```
while ( condition ){
    statement
}
```

Solange condition true ist, wird statement ausgeführt.

```
do{
    statement
}while ( expression );
```

Statement wird ausgeführt, von da an wie `while`

Sprunganweisungen in C++

`goto identifier;` →springt zu *identifier* der an einer Stelle mit `identifier:` gesetzt wurde
`break;` →springt aus der **umschliessenden** Iterationsanweisung heraus
`continue;` →überspringt den Rest des Statements

Scope/Gültigkeit/Sichtbarkeit

Die Deklaration einer Variablen ist nur innerhalb eines Blockes sichtbar. Dieser *potenzielle* Gültigkeitsbereich heisst scope und wird mit geschweiften Klammern gekennzeichnet.

Aufbau einer Funktion

```
T funktionsname (T1 arg1, T2 arg2…){
    Block //(von statement)
}
```

T ist dabei der Rückgabetyp (oder `void`). Falls T `void`, *kann* sie – zum Abbruch – ein `return;` enthalten, falls T nicht `void`, **muss** die Funktion ein `return` enthalten.

Funktionsargumente treten **nie** lokal auf, also nie in andern Funktionen etc.

Pre- und Postcondition

Beschreibung, was die Funktion «macht»
`//PRE: …` → Was muss bei Funktionsaufruf gelten?
→Spezifiziert Definitionsbereich der Funktion.
`//POST: …` → Was muss bei Funktionsaufruf gelten?
→Spezifiziert Wert und Effekt des Funktionsaufrufes.
PRE und POST können mit Assertions geprüft werden

Gültigkeit und Forward Declarations

Funktionen können erst nachdem sie im Programm stehen auch aufgerufen werden (vgl. Scope bei Variablen). Falls sie vorher benötigt werden → Forward Declaration: Die Funktion wird oberhalb hingeschrieben (ohne `{block}`, dafür mit Semikolon) und weiter unten deklariert.

Inkludieren einer Datei

`Z.B. #include "bibliothek.h"`

Call by Reference / Call by Value

`T function(T arg, …);` → Für die Funktion wird eine lokale Kopie des Arguments gemacht → Änderungen an `arg` **beeinflussen** das ursprünglich übergebene Objekt **nicht**.
`T function (T& arg, …);` → Der Funktion wird eine Referenz auf das Objekt übergeben → Änderungen an `arg` **beeinflussen** das ursprünglich übergebene Objekt **direkt**.

Return by Value/ by Reference

Auch der Rückgabetyp einer Funktion kann ein Referenztyp sein. In diesem Fall ist der Funktionsaufruf selbst ein L-Wert.
Achtung: Wenn man eine Referenz erzeugt, muss das Objekt, auf das sie verweist, mindestens so lange „leben” wie die Referenz selbst.

Konstanten

Konstanten werden mit dem Präfix `const` gekennzeichnet. Der Compiler überprüft, dass Konstanten nicht verändert werden. Variablen, die ihren Wert nicht ändern **immer** `const` setzen!

```
const int speed_of_light = 299792458;
```

Const Referenzen

`T value = something; //` Variablendeklaration
`T& r = value; //` r ist ein Lese-Schreib-Alias
`const T& r = value; //` r ist ein Lese-Alias
Wenn Typ T gross: Beim übergeben Argumenttyp `const T&` (call by read-only reference) verwenden→ bessere Laufzeit

Templates

Templates erlauben die Angabe eines Typs als Argument (gleiche Funktion für double und int)

```
template<class T(, class U)>
```

mit default values:

```
template<class T=int(, class U=double)>
```

Funktion dazu:

```
T getMax (T a, U b) {
  return (a>b?a:b);
}
```

Funktionsaufruf:

```
getMax <int, double> (a, b);
```

oder einfacher `getMax(a, b);`

Felder

Erlauben wahlfreien Zugriff. D.h. man kann zu jedem Zeitpunkt über einen Feldindex genau ein bestimmtes Element aufrufen.

Array

`int a[5]; //` Feld mit 5 Elementen uninitialisert
`int a[5] = {4, 3, 5, 2, 1}; //` Liste
`int a[] = {4, 3, 5, 2, 1}; //` Die fünf Elemente von a werden mit einer Liste initialisiert.
Einzelne Elemente werden über deren Indizes abgerufen. Z.B. `int b = a[2] // b = 5`
C++ überprüft von selbst nicht ob alle Elementzugriffe gültig sind (z.B. über Listengrenzen hinaus).

Vektoren

```
#include <vector>
std::vector<T> vec (n, T0);
```

T = Elementtyp, Initialisierung mit n Elementen Initialwert T0
Wird ein zusätzliches Element benötigt:
`vec.pushback(new Element)`

String als char-Felder

Char-Felder: `char text[] = "bool" // gleich char text[] = {'b','o','o','l','\0'}`

Strings

```
std::string text = "bool";
```

benötigt `#include<string>`

String ist ein Feld als Vektor von char-Elementen.

Darum: `string.at(int stelle)`, `string.lenght()`, Vergleich mit `==`, `std::string text(n,'a')` //Initialisierung mit variabler Länge

Mehrdimensionale Felder / Vektoren

Initialisierung:

`int a[n][m] //` nxm Feld uninitialisierte Elemente

`int a[][3] = { {2,4,6},{1,3,5} }`

Vektoren: mehrere Dimensionen als Vektoren von Vektoren: `std::vector<std::vector<int> > a(n, std::vector<int>(m));` !Lücke bei <int> >

Felder als Funktionsargumente

```
function(const int (&v)[3]){…}
function(const int (&m)[3][3]){…}
function(const std::vector<int>& v) {…}
function(const
std::vector<std::vector<int> >& m) {…}
```

`Call-by-Reference` wenn nicht anders nötig und `const` wenn nur Lesezugriff nötig.

Lindenmayer-Systeme

Bestehen aus einem Alphabet (z. B. { `F , + , −` }), einer Produktion (z. B. `P(F) = F+F`, od. `P(+) =-F+`) und einem Startwert (z. B. `s = F +`). Durch mehrere Iterationen über die Produktion wird ein immer längeres Wort gebildet. Diese werden nach gewissen Regeln dargestellt (z.B. **oft** gehende Turtle mit `F` = vorwärts, `+/-` = Drehung um +- 90 Grad)

Pointer, Iteratoren und Container

Wahlfreier Zugriff ist selten nötig und verhältnismässig langsam → sequenzieller Zugriff

Dafür benötigen wir Pointer:

Funktionieren wie Referenzen, können aber das referenzierte Objekt ändern.

`int* p;` Variable p ist Zeiger auf ein int.

Adress- und Dereferenz-Operator

`& lval`

Der Adress-Operator `&` liefert als R-Wert einen Zeiger vom Typ T* auf das Objekt an der Adresse von lval.

`*rval`

Der Dereferenz-Operator `*` liefert als L-Wert den Wert des Objekts an der, durch rval repräsentierten Adresse.

Konversion Felder zu Zeiger

```
int a[5];
int* begin = a; // begin zeigt auf a[0]
```

Längeninformation geht verloren.

Iteration mit Pointer

```
int a[5] = {3, 4, 6, 1, 2};
for (int* p = a; p < a+5; ++p);
```

Integer Addition

Addiert/Subtrahiert man ein Integer zu einem Pointer einer Liste, wird der Pointer um so viele Elemente weiterverschoben.

Felder mit Pointer übergeben

Bei der Übergabe des Pointers auf das erste Element geht die Längeninformation verloren. Darum Zeiger `begin` auf erstes Element und `end` auf **hinter** das letzte Element. [begin, end) bezeichnet die Elemente des Feldausschnitts und Feld ist leer wenn begin==end

Nullpointer

Nicht dererferenzierbar `int* iptr = nullptr;`

Pointer-Subtraktion

Pointer – Pointer gibt die Anzahl Elemente dazwischen.

`(expr*).func();` ist dasselbe wie `expr->func();`

Const und Pointer v.r.n.l.

`int const a;` a ist eine konstante Ganzzahl

`int const* a;` a ist ein Zeiger auf eine konstante Ganzzahl <-->

`int* const a;` a ist ein konstanter Zeiger auf eine Ganzzahl

`int const* const a;` a ist ein konstanter Zeiger auf eine konstante Ganzzahl

`Const` ist nicht absolut bei Zeigern, wenn er auf ein nicht konstantes Objekt zeigt!

Container = Behälter (Feld, Vektor,. . .) für Elemente

Traversierung = Durchlaufen des Containers

Durchlaufen entweder sequenziell oder als wahlfrei.

Iteratoren

Da Länge von z.B. Vektoren variabel je nach Typ, brauchen wir Iteratoren. Ist kein Zeiger, verhält sich aber ähnlich. `vec.begin()` bzw. `vec.end()` zeigt auf das erste bzw. hinter das etzte Element.

```
for (std::vector::const_iterator it =
v.begin(); it != v.end(); ++it)
```

Iteratoren können nur auf Gleichheit geprüft warden.

Set

`std::set<T>` für eine Menge mit Elementen vom Typ T, wobei jedes Element nur einmal vorkommt.

Rekursion

Wie in der Mathematik: die Funktion erscheint in ihrer eigenen Definition. «Funktion der Funktion»

Braucht immer Terminierung

Rekursion und Iteration

Rekursion kann immer simuliert werden durch: Iteration (Schleifen), expliziten „Aufrufstapel" (z.B. Feld). Rekursion oft einfacher aber ineffizienter.

EBNF (Extended Backus Naur Form)

EBNF definiert woraus einzelne Ausdrücke bestehen:

```
fac = num | "(" expr ")" | "-" fac.
term = fac { "*" fac | "/" fac }.
```

{} ist eine optionale Repetition

Structs und Classes

Definieren einen neuen Typen

Structs

`struct T{T1 name1; T2 name2; };`

`T` = Name des neuen Typs, `T1`, `T2`… bzw. `name1`, `name2` Typen bzw. Namen der Membervariablen

Member und -Zugriff

`expr.name` gibt die Membervariable von `expr` zurück

Initialisierung und Zuweisung

```
my_struct m; // Default-Initialisierung
my_struct s = {1, "Hello"}; // Normalfall
m = s // Zuweisung: Member-Variablen von m
```

werden jeweilige die Werte von s zugewiesen

`m = s` Achtung bei Pointern: Objekt dahinter wird nicht neu generiert sondern Pointer zeigt auf das Erste.

Funktions- und Operatorüberladung

Funktionsüberladung erlaubt es, mehrere Funktionen des gleichen Namens zu definieren und zu deklarieren. Die „richtige" Version wird aufgrund der Signatur (Argumenttypen) der Funktion ausgewählt.

Alle Operatoren werden mit `operatorOP` überladen: OP ist dabei das «Zeichen»

binäre Operatoren (+, -, *, /, %) out-class

```
T operator+ (T a, T b){…}
```

Unäre Operatoren (+, -, *, /, %) meist in-class

```
T operator- (T a){…}//out-class: z.B.
negieren von Werten int a= 5; -a;→a=-5
```

Vergleichsoperatoren ==, !=, <, >, <=, >=

```
bool operator== (T a, T b){…}
```

Operator +=, -=, *=, /= und %=

```
T& operator+= (T& a, T b) {…}
```

Prä- und Postfix ++ und - -

```
T& operator++ (){…} // für prefix ++, also ++c
T operator++ (int){…} // für postfix ++, also
```

c++, das `(int)` hat nur kennzeichnende Funktion.

Ausgabeoperator

```
std::ostream& operator<< (std::ostream&
out, T a){
    return out (<< evtl etwas aus a);
}
```

Eingabeoperator

```
std::istream& operator>> (std::istream&
in, T& a) {
    return in >> a.name1 >> a.name2;
}
```

Zur Erinnerung: Grosse Objekte übergibt man als Const-Referenz

Classes

Variante von Structs mit Datenkapselung.

Per default alle Funktionen und Variablen privat.

Änderung durch Annotation `private:` nur in der Klasse, `public:` von aussen sichtbar und `protected:` sichtbar für erbende Klassen

Auch bei Klassen funktioniert Forward-Declaration!

Member Funktion

In und out-class definierbar: In class wie sonst auch (schlecht für Bibiliotheken), out-class folgendermassen:

```
T classname::functionname(args){…};
```

Können überladen werden, also in der Klasse mehrfach – aber mit verschiedener Signatur – vorkommen.

Falls `const` am Ende der Funktion: Die Funktion darf keine Member-Variable der Klasse verändern.

Constructor und Destructor

Constructor initialisiert die Klasse, wird vom Compiler automatisch ausgewählt: eine spezielle Funktion mit Namen der Klasse. Mind. ein Constructor muss existieren: in-class: `classname(args){…}`, out-class: `classname::classname(args){…}`

Destructor: Falls vor dem Löschen des Objekts noch etwas getan werden muss (z. B. dynamisch allozierte Variablen löschen) hier drin. Wird folgendermassen dargestellt in-class: `~classname(){…}`, out-class: `classname::~classname(){…}`

`this` ist in-class ein Pointer auf sich selbst

Beispielklasse

```
class rational{
   public:
   rational(int n, int d):num
            (n),den(d){…};
   rational (int n): num(n),den(1){…};
   private:
   int num, den;
};
```

Initialisierung:

```
rational r = rational (1, 2);
```

oder `rational r (1, 2);`

Konversion:

`rational r = 2 //` implizite Konversion

Dynamischer Speicher

Verkettete Listen

Kein zusammenhängender Speicherbereich und kein wahlfreier Zugriff, Jedes Element „kennt" nur seinen Nachfolger, Einfügen und Löschen beliebiger Elemente ist einfach, auch am Anfang der Liste, allerdings muss daran gedenkt werden, alle Pointer anzupassen.

`new, delete expr, delete[] expr`

`new T`: Neues Objekt vom Typ T wird im Speicher angelegt, `delete expr` bzw. `delete[] expr`: Objekt bzw. Feld wird gelöscht, Speicher wird wieder freigegeben.

Zu jedem `new` gibt es ein passendes `delete`!

Copy-Constructor

Der Copy-Konstruktor einer Klasse T ist der eindeutige Konstruktor mit Deklaration, wird automatisch aufgerufen, wenn Werte vom Typ T mit Werten vom Typ T initialisiert werden:

`T x = t; od. T x (t); // t vom Typ T`

wenn er fehlt: automatische memberweise Initialisierung. Z.T. Problematisch, da für Pointern die Adresse kopiert wird und nicht ein neuer Pointer auf eine Kopie eines Objekts zeigt.

Zuweisungsoperator

Überladung von `operator=` als Memberfunktion: wie Copy-Konstruktor aber: Freigabe des Speichers für den „alten" Wert, Prüfen auf Selbstzuweisungen

Bibliothek

class.h: Definition des Structs (bzw. der Klasse), Funktionsdeklarationen (der Constructor)

class.cpp: Deklaration der Funktionen. Überladung von Operatoren und Funktionen des Structs (der Klasse).

class.cpp muss class.h inkludieren.

int → unsigned int: $x \to x$; $-x \to 2^{32}-x$ wobei: $x>0$

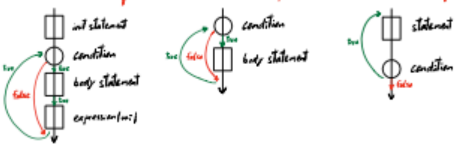
int → float, double: Zahl bleibt unverändert (bei sehr großen Zahlen kann es kleine Fehler geben)

float, double → int: Zahl wird abgerundet

Ganze Zahlen & Fließkommazahlen

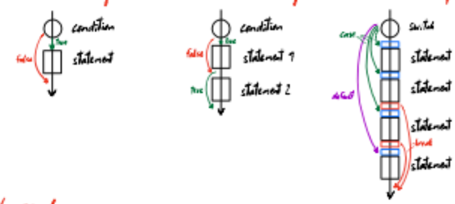
1 → Typ: int; Wert: 1
1u → Typ: unsigned int; Wert: 1
1.0 → Typ: double; Wert: 1 (64 Bit)
1.0f → Typ: float; Wert: 1 (32 Bit)

for-Anweisung / while-Anweisung / do-Anweisung



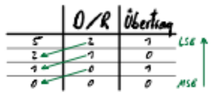
| do-Schleife endet mit Semikolon! (do{...} while(...);)

if-Anweisung / if-else-Anweisung / switch-Anweisung



Umrechnungen

dezimal → binär:



binär → dezimal: $101_{(2)} = 1\cdot2^2+0\cdot2^1+1\cdot2^0 = 5_{(10)}$

Fließkommazahlensysteme

Basis $\beta \geq 2$
Präzision (= Stellenzahl) $p \geq 1$ (inkl. der Zahl vor dem Komma)
kleinster Exponent e_{min}
größter Exponent e_{max}

$F(\beta, p, e_{min}, e_{max})$

Normalisierte Darstellung: $F^*(\beta, p, e_{min}, e_{max})$ (normalisiert)

- $\pm d_0.d_1\dots d_{p-1} \times \beta^e$ wobei $d_0 \neq 0$ (Vor dem Komma darf nur eine Stelle stehen!)

- Anzahl pos. Werte: $(\beta-1)\cdot\beta^{p-1}\cdot(e_{max}-e_{min}+1)$
- Anzahl Werte: (Anzahl pos. Werte) $\cdot 2$
- Größte Zahl: $(\beta^1-1)\cdot\beta^{e_{max}-p+1}$
- Kleinste positive Zahl: $\beta^{e_{min}}$

Fließkomma-Richtlinien

i) Teste keine gerundeten Fließkommazahlen auf Gleichheit!
ii) Addiere keine zwei Zahlen sehr unterschiedlicher Größe!
iii) Subtrahiere keine zwei Zahlen sehr ähnlicher Größe!

Funktionen

- kapseln häufig gebrauchte Funktionalität
- Änderung ihrer Werte haben keinen Einfluss auf die Werte der Aufrufargumente! (außer bei: Referenzen)

| Funktionsdefinitionen dürfen nicht lokal (in Blöcken oder anderen Funktionen) auftreten!

Vorbedingung: so schwach wie möglich (großer Definitionsbereich)
Nachbedingung: so stark wie möglich (detaillierte Aussage)

Namespace

- Mit Namespaces kann man Funktionen, Typen, ... katalogisieren
- Bsp.:

```
namespace ifmp{          //namespace called ifmp
    void output_func(){std::cout<<"Hi";}
}
int main(){
    ifmp::output_func(); //use output_func from namespace ifmp
    return 0;}
```

Referenztypen

T& → T-Referenz; kann nur mit L-Value initialisiert werden (zugrundeliegender Typ)

Pass by Reference/Value: Argument wird mit Adresse/Wert des Aufrufsarguments initialisiert (Alias/Kopie)

| Das Objekt, das auf eine Referenz verweist, muss mind. so lange "leben" wie die Referenz selbst!

| Referenzen haben keine Adresse! (Pointer haben eine!)

Const-Referenzen

- kann mit L- & R-Value initialisiert werden
- Bsp.: T ist kein Referenztyp:

```
const int n=5;
int& i=n; //error: const-qualification is discarded
i=6;
```

- Bsp.: T ist Referenztyp:

```
int n=5;
const int& i=n; //i: Lese-Alias von n
int& j=n;       //j: Lese-Schreib-Alias von n
i=6;            //error: i ist Lese-Alias
j=6;            //valid: n bekommt Wert 6
```


©Emanuel Pyllik / ✉: pyllike@student.ethz.ch

Konstruktoren

spezielle Memberfunktionen einer Klasse, die immer den Namen der Klasse tragen

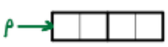
```
T(int variable1, double variable2){
  : i(variable1), d(variable2)
}
```

Default-Konstruktor: wird bei leerer Argumentliste aufgerufen (Bsp.: rational r;)

Dynamische Datenstrukturen

Neuen Speicher allozieren

p = new T[expr] (zugrundeliegender Typ (default); Typ int, Wert n)



Pointer T* → Zeiger auf T

- Wert eines Zeigers auf T ist Adresse eines Objekts vom Typ T

Adress-Operator: &expr (L-Wert vom Typ T)
- Wert: Adresse des Objekts "expr"
- Typ: Zeiger T* (vom Typ T)

Dereferenz-Operator: *expr (R-Wert vom Typ T*)
- Wert: Wert des Objekts an durch expr bestimmten Adresse
- Typ: T

Null-Zeiger: nullptr
- gibt an, dass (noch) auf kein Objekt gezeigt wird
- kann nicht dereferenziert werden

Zeiger-Arithmetik:
- $*(p+i) = p[i]$

Arrays in Funktionen:

- begin: Zeiger auf das erste Element
- end: Zeiger hinter das letzte Element
- [begin, end) bezeichnet die Elemente des Array Ausschnitts
- [begin, end) ist leer, wenn begin == end

Konstante Pointer:
- int const p; p ist eine konstante Ganzzahl
- int const* p; p ist ein Zeiger auf eine konstante Ganzzahl
- int* const p; p ist ein konstanter Zeiger auf eine Ganzzahl
- int const* const p; p ist ein konstanter Zeiger auf eine konstante Ganzzahl

| Der Wert an einer Adresse kann sich ändern, auch wenn ein const-Zeiger diese Adresse speichert!

Linked List (Verkettete Liste)

- Kein zusammenhängender Speicherbereich und kein wahlfreier Zugriff
- Jedes Element zeigt auf seinen Nachfolger

Container und Iteratoren

Container ("Ansammlungs"-Datenstrukturen)

- std::unordered_set<T>
(un)geordnete, duplikatfreie Zusammenfassung von Elementen

Iteratoren T::iterator (zugrundeliegender Container (z.B. std::vector<int>))

Iteration über Array: for(int* it = begin; it != end; ++it){ std::cout << *it;} //aktuelles Element ausgeben

Iteration über verkettete Liste: for(llnode* n = this->head; n != nullptr; n = n->next){ std::cout << n->value;} //aktuelles Element ausgeben

| Jeder Container implementiert seinen eigenen Iterator! (der Name des Containers ist C)

- it = C.begin() — Iterator aufs erste Element
- it = C.end() — Iterator hinters letzte Element
- *it — Zugriff aufs aktuelle Element
- ++it — Iterator um ein Element verschieben

Dynamische Datentypen und Speicherverwaltung

Speicher Löschen

delete expr → Objekt wird dekonstruiert & Speicher freigegeben (Zeiger vom Typ T*, der auf ein vorher mit new erzeugtes Objekt zeigt)

delete[] expr → Feld wird gelöscht & Speicher freigegeben (Zeiger vom Typ T*, der auf ein vorher mit new erzeugtes Feld zeigt)

Dynamischer Datentyp (Typ, der dynamischen Speicher verwaltet)

Mindestfunktionalität:
- Konstruktoren
- Dekonstruktor
- Copy-Konstruktor
- Zuweisungsoperator

Dreierregel: definiert eine Klasse eines davon, so muss sie auch die anderen zwei definieren

Anhang

Potenztabelle

x	2^x	2^{-x}	16^x
0	1	1	1
1	2	0,5	16
2	4	0,25	256
3	8	0,125	4096
4	16	0,0625	65536
5	32	0,03125	1048576
6	64	0,015625	-
7	128	-	-

Bad expressions

(a+b)*(a++)

| Ergebnis hängt von Auswertungsreihenfolge vom Compiler ab! => nicht definiert

©Emanuel Pyllik / ✉: pyllike@student.ethz.ch