Initialisierung vs. Zuweisung:

- im Kontext einer Objektdeklaration: Initialisierung

```
X x = something; // initialize
```

- nicht im Kontext einer Objektdeklaration: Zuweisung

```
x = something; // assign !
```

```
class X {
  const int c;
public:
  X(int i): c(i) {} // ok, aber
  // X(int i) {c=i;} // falsch
};
```

☝ Prefer initialization !

Initialisierung vs. Zuweisung:

```
#include <iostream>

class A {
public:
   A(int i){ std::cout<<"A("<<i<<")\n"; }
};

class B {
   A myA;
public:
   B (int i) { std::cout<<"B("<<i<<")\n"; }

};

int main() { A a(1); B b(2); } // valid C++ ?????
```

Initialisierung vs. Zuweisung:

Prefer initialization !

```
#include <iostream>

class A {
public:
   A(int i){ std::cout<<"A("<<i<<")\n"; }
};

class B {
   A myA;
public:
   B (int i) { std::cout<<"B("<<i<<")\n"; }

};

int main() { A a(1); B b(2); }
```

```
Error init.cpp 11: Cannot find default constructor to
initialize member 'B::myA' in function B::B(int)
```

Initialisierung vs. Zuweisung:

Prefer initialization !

```
#include <iostream>

class A {
public:
   A(int i){ std::cout<<"A("<<i<<")\n"; }
};

class B {
   A myA;
public:
   B (int i) { myA = i; std::cout<<"B("<<i<<")\n"; }

};

int main() { A a(1); B b(2); }
```

```
Error init.cpp 11: Cannot find default constructor to
initialize member 'B::myA' in function B::B(int)
```

Initialisierung vs. Zuweisung:

Prefer initialization !

```
#include <iostream>

class A {
public:
   A(int i = 0 ){ std::cout << "A("<<i<<")\n"; }
};

class B {
   A myA;
public:
   B (int i) { myA = i; std::cout << "B("<<i<<")\n"; }

};

int main() { A a(1); B b(2); }
```

```
$ init
A(1)
A(0)  ?
A(2)
B(2)
```

Initialisierung vs. Zuweisung:

Prefer initialization !

```
#include <iostream>

class A {
public:
    A(int i){ std::cout << "A("<<i<<")\n"; }
};

class B {
    A myA;
public:
    B (int i): myA(i) { std::cout << "B("<<i<<")\n"; }

};

int main() { A a(1); B b(2); }
```

```
$ init
A(1)
A(2)
B(2)
```

Initialisierung vs. Zuweisung:

```
#include <iostream>

class A {
public:
    A(int i){ std::cout << "A{"<<i<<"}\n"; }
};

class B {
    A myA;
public:
    B (int i): myA{i} { std::cout << "B{"<<i<<"}\n"; }

};

int main() { A a{1}; B b{2}; }
```

☝ even better: uniform initialization !

```
$ init
A{1}
A{2}
B{2}
```

☝ C++ idiom: Resource Acquisition Is Initialization (*

```
void doDB() { // from Steven C. Dewhurst: C++ Gotchas (gotcha #67)
    lockDB();
    // do stuff with database ... but could throw !?
    unlockDB();
}

void doDB() {
    lockDB();
    try {// do stuff with database ...
    }
    catch ( ... ) { unlockDB(); throw; } // ugly
    unlockDB();
}
```

(* of an object !

C++ idiom: Resource Acquisition Is Initialization

```
// better:
class DBLock {
   public:
        DBLock() { lockDB(); }
        ~DBLock() { unlockDB(); }
};

void doDB() {
   DBLock lock;
   // do stuff with database ...
}
```

Fallen:

```
// NOT: DBLock lock();
// NOT: DBLock();
```

C++ idiom: Resource Acquisition Is Initialization

```
struct X {
        X() { cout<<"X()\n"; }
        ~X() { cout<<"~X()\n"; }
};

struct Xpointer { // a (not very) smart pointer
        X* pointer;
        Xpointer(X* p): pointer(p){}
        ~Xpointer(){delete pointer;}
};

struct Y {
        Xpointer p;
        Y(int i) try : p(new X)
        { if (i) throw "huhh"; }
        catch(...)
        { cout<< "catched local\n";}
        ~Y() {}
};

int main() try {
        cout<<"sizeof(Y)="<<sizeof(Y)<<endl;
        Y y0(0);
        Y y1(1);
}
catch(...) { cout<<"catched final\n";}
```

```
#include <iostream>
using std::whatever;
```

```
sizeof(Y)=4
X()
X()
~X()
catched local
~X()
catched final
```

C++ idiom: Resource Acquisition Is Initialization

C++11

```
#include <iostream>
#include <memory>

struct X {
    X() { std::cout<<"X()\n"; }
    ~X() { std::cout<<"~X()\n"; }
};

struct Y {
    std::unique_ptr<X> p;
    Y(int i) try : p(new X)
    { if (i) throw "huhh"; }
    catch(...)
    { std::cout<< "catched local\n";}

    ~Y() {}
};

int main()
    try {
        std::cout<<"sizeof(Y)="<<sizeof(Y)<<std::endl;
        Y y0(0);
        Y y1(1);
    }
    catch(...) { std::cout<<"catched final\n"; }
```

```
sizeof(Y)=8
X()
X()
~X()
catched local
~X()
catched final
```

```
#include <iostream>
using std::whatever;
```

C++ idiom: Resource Acquisition Is Initialization

```
class Trace { // C++ Gotchas, dito #67
public:
    Trace (const char* msg): m_(msg) {cout << "Entering " << m_ << endl;}
    ~Trace() {cout << "Exiting " << m_ << endl;}
private:
    const char* m_;
};
Trace a("global");
void foo(int i) {
    Trace b("foo");
    while (i--) { Trace l("loop"); /* ... */ }
    Trace c("after loop");
}
int main() { foo(2); }
```

```
$ t
Entering global
Entering foo
Entering loop
Exiting loop
Entering loop
Exiting loop
Entering after loop
Exiting after loop
Exiting foo
Exiting global
```

```
#include <iostream>
#include <ctime>
using std::whatever;
```

C++ idiom: Resource Acquisition Is Initialization

```
class Timer {
   long start, stop;
   void report()
       {cout<<(stop-start)/1000000.0<<"s"<<endl;}
public:
   Timer():start(clock()){}
   ~Timer(){ stop=clock(); report();}
};
```

```
#include <iostream>
#include <chrono>
```

C++ idiom: Resource Acquisition Is Initialization

```
class Timer { // conforms to C++11
    std::chrono::steady_clock::time_point start;
    std::string what;
public:
    Timer(std::string s): start(std::chrono::steady_clock::now()), what(s) {}
    ~Timer() {
        auto duration = std::chrono::steady_clock::now() - start;
        std::cout << what+":\t" <<
        std::chrono::duration_cast<std::chrono::milliseconds>(duration).count()
        << " ms" << std::endl;
    }
};
```

- Klassen können auch sogenannte static Member enthalten, diese werden nur einmal pro Klasse angelegt !
- `static` Memberfunktionen dürfen (implizit) nur auf static Memberdaten zugreifen, (sie haben keinen `this`-Zeiger!)
- `static` Memberdaten sind nicht Teil des Objekt-Layouts
- `static` Memberdaten sind (einmalig) zu initialisieren !

2. Klassen in C++

```
$ s
Es gibt jetzt 3 A-Objekte
```

```
class A {
   static int count;
public:
   static int c(){ return count; }
   static const double A_specific_const; // NOT HERE = 123.456;
   A() {count++;}
   A(const A&) {count++; /* and copy */} // Kopien mitzählen !
   ~A(){count--;}
} a1, a2, a3;
int A::count = 0; // hier erst definiert !
const double A::A_specific_const = 123.456; // dito
int main() {
   double x = A::A_specific_const; // class access
   // A::A_specific_const = 1.23; // Fehler: const !
   cout << "Es gibt jetzt "<< a1.c()<<" A-Objekte\n";
// a1.count ist private,auch a2.c() oder a3.c() oder A::c() möglich
}
```

- neben den traditionellen C-Zeigern gibt es in C++ auch spezielle Zeigertypen für Zeiger auf Member(-daten und -funktionen)

✪

```
class X { public: int p1,p2,p3; };
void foo() {
  X x; X* pp=&x;      // ein C-Zeiger auf ein X
  int X::*xp=&X::p2; // xp ist ein Zeiger auf ein int in X
// xp = &x.p2;
// error: bad assignment type: int X::* = int *
  int *p;
// p = &X::p2;
// error: bad assignment type: int * = int X::*
  p = &(x.*xp); // ok, ohne Klammern falsch: (&x).*xp
  pp->*xp = 1; } // .* und ->* sind neue Operatoren
```

```
class Y {
public:
  void f1(){cout<<"Y::f1()\n";};
  void f2(){cout<<"Y::f2()\n";};
  static void f3(){cout<<"static Y::f3()\n";}
  typedef void (Y::*Action)();
  void repeat(Action=&Y::f1, int=1);//...(void(Y::*)(),int)
};
void Y::repeat (Action a, int count) {
  while (count--) (this->*a)();
}
int main() {
  Y y; Y* pp=&y;
  void (Y::*yfp)();
  // Zeiger auf Memberfkt. in Y mit Signatur void->void
```

```
yfp=&Y::f1; // nicht yfp =Y::f1 !(trotz vc++6.0, bcc32, icc)
// yfp();
// object missing in call through pointer to memberfunction
(y.*yfp)(); // Y::f1()
yfp=&Y::f2;
(pp->*yfp)(); // Y::f2()
// yfp=&Y::f3;
// bad assignment type: void (Y::*)() = void (*)()static
// aber:
void (*fp)()=&Y::f3;
fp(); // besser (*fp)();
y.repeat(yfp, 2);
}
```

```
$ mp
Y::f1()
Y::f2()
static Y::f3()
Y::f2()
Y::f2()
```

Vererbung: Grundprinzip von OO

- Übernahme von Eigenschaften aus einer Klasse
- Erweiterung / Modifikation

Beispiel: ein Stack mit Buchführung

```
class CountedStack : public Stack // IST EIN STACK
{
   int min, max, n, sum; // zusätzliche Attribute
public:
   CountedStack(int dim = 100);
   void push (int i); // redefined !
   int minimum(); // neu
   int maximum(); // neu
   double mean(); // neu
   double actual_mean();// neu
// pop, empty, full aus der Basisklasse !
};
```

2. Klassen in C++ back -->

```
CountedStack::CountedStack(int dim):Stack(dim),n(0),sum(0){}

void CountedStack::push(int i) {
   sum+=i;
   if (!n++) { min = max = i; }
   else { min = (i<min) ? i : min; max = (i>max) ? i : max; }
   Stack::push(i); // use base functionality NOT push(i)
}

double CountedStack::actual_mean() {
   if (top) { int s=0;
       for (int i=0; i<top; i++) s += data[i];
       return double(s)/top; // direct access to base members
   } else std::exit(-4);
}
```