- Wann immer Objekte entstehen, läuft automatisch ein (passender) Konstruktor !
- Wann immer Objekte verschwinden, läuft automatisch der Destruktor !
- Klassen ohne nutzerdefinierten Konstruktor/Destruktor besitzen implizit
  - den sog. *default constructor* `X (){}`
  - den sog. *default copy-constructor* `X (const X&){ ... }` und
  - den sog. *default destruktor* `~X() {}`

memberweise Kopie !

- sobald nutzerdefinierte Konstruktor-Varianten vorliegen, gibt es nur noch den impliziten Copy-Konstruktor (wenn dieser nicht auch explizit definiert wird)

- Jedes Objekt enthält seine eigene Realisierung der Memberdaten (NICHT der Memberfunktionen!)
- Die Identität eines Objektes ist mit seiner Adresse verbunden !

```
bool Any::same(Any& other){
        return this == &other;
}
```

?

Beispiel: Jedes `Stack`-Objekt hat das folgende Layout unabhängig davon, wie es entstanden ist !

kein overhead durch Meta-Daten !

- es sind auch sog. unvollständige Klassendeklarationen erlaubt, von einer solchen Klasse können jedoch bis zu ihrer vollständigen Deklaration lediglich Zeiger & Referenzen benutzt werden:

```
class B;
class A {B * my_B; ....}; // oder ... class B* my_B;
class B {A * my_A; ....};
```

- strukturell identische Klassen mit verschiedenen Namen bilden verschiedene Typen (es gibt jedoch die Möglichkeit, nutzerdefiniert Kompatibilität herbeizuführen s.u.):

```
class X { public: int i; } x0;
class Y { public: int i; } y0;
X x1 = y0; Y y1 = x0; // beides falsch !!!
x0 = y0; y0 = x;      // beides falsch !!!
```

- Konstruktorparameter sind beim Anlegen von Objekten (geeignet) anzugeben, d.h es muss einen entsprechenden Konstruktor geben

```
// direct initialization:
X x0;                    // needs X::X();
X x1(1);                 // needs X::X(int);
X x2 = X(2,0);           // needs X::X(int,int);
X *pb = new X (5,true);  // needs X::X(int, bool);
X x3(1, "zwei", '3'); // needs X::X(int,[const] char*,char);

// copy initialization:
X x3 = 1;    // X tmp(1); X x3(tmp); ggf. elision
```

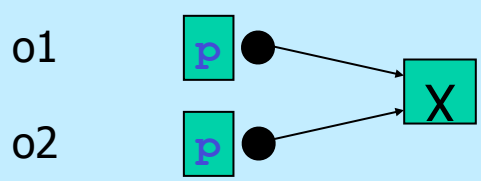
Copy-Konstruktoren

```
X::X(const X&); // kanonische Form !
```

shallow copy

(default copy ctor)

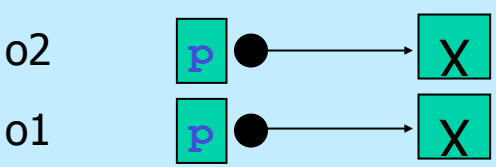
```
class SC {
   X* p;
public:
   SC(): p(new X) {}
};
SC o1;
SC o2=o1;
```



deep copy

(nutzerdefinierter copy ctor)

```
class DC {
   X* p;
public:
        DC(): p(new X) {}
        DC(const DC& src)
        : p(new X) {...copy X };
};
DC o1;
DC o2=o1;
```



- Konstruktoren können auch mit einem function try block implementiert werden, auch wenn ein passender handler vorliegt, wird die Ausnahme **immer** re-thrown !!!

```
struct Y {
   X* p;
   Y(int i) try : p(new X)
   { if (i) throw "huhh"; }
   catch(...)
   { /* delete p; NOT ALLOWED !!! */
     /* throw "huhh"; implicitly  */}
   ~Y() { delete p; }
};
```

15.3 (10): Referring to any non-static member or base class of an object in the handler for a function-try-block of a constructor or destructor for that object results in undefined behavior.

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)
```

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

```
#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;
    }
};
```