# initializer lists


(nicht für Referenzen ... unklar warum ...)

```
class Y{} y ;

Y &ref (y) ;  // ok
Y &ref {y} ;  // Fehler: g++, MS VS C++11
```

# preventing narrowing

```
int x = 7.3; // Ouch!
void f(int);
f(7.3); // Ouch!
```

mit {} Initialisierung nicht:

```
int x1 = {7.3}; // error: narrowing
double d = 7;
int x2{d}; // error: narrowing (double to int)
char x3{7};
     // ok: even though 7 is an int, this is not narrowing
vector<int> vi = { 1, 2.3, 4, 5.6 };
     // error: double to int narrowing
```

# delegating constructors

Wenn Konstruktoren ähnliches tun:

```
class X {
 int a;
 validate(int x) {
  if (0<x && x<=max) a=x; else throw bad_X(x);
 }
public:
 X(int x) { validate(x); }
 X() { validate(42); }
 X(string s) {
  int x = lexical_cast<int>(s); validate(x);
 }
};
```

- Schlecht: Lesbarkeit, Fehleranfälligkeit, Wartbarkeit

# delegating constructors

Einen Konstruktor mit Hilfe eines anderen implementieren:

```
class X {
 int a;
public:
 X(int x) {
  if (0<x && x<=max) a=x; else throw bad_X(x);
 }
 X() : X{42} { }
 X(string s) :X{lexical_cast<int>(s)} { }
};
```

dann darf nach : **NUR** der andere Ctor stehen (keine weitere inits)!

# delegating constructors

Sieht auf den ersten Blick trivial aus.

- der zweite offenbart ein semantisches Problem

```
class X {
 ...
 X() : X{42} { if (something) throw "oops"; }
};
```

bisher galt in C++98:

- tritt in einem Konstruktor eine Exception auf, ist das Objekt NIE entstanden - der Destruktor wird auch nicht gerufen
- wenn ein Konstruktor erfolgreich (ohne Exception) beendet wurde, ist ein Objekt entstanden, der Destruktor muss gerufen werden (ggf. implizit)

nun gilt in C++11:

- wenn ein (Teil-)Konstruktor erfolgreich (ohne Exception) beendet wurde, ist ein Objekt entstanden, der Destruktor muss gerufen werden (ggf. implizit)

DEMO: delegate

# in-class member initializers

In C++98 kann man nur static const members integraler Typen in der Klasse mit einem konstanten Ausdruck initialisieren.

```
int var = 7;
class X {
  static const int m1 = 7; // ok
  const int m2 = 7;        // error: not static
  static int m3 = 7;       // error: not const
  static const int m4 = var; // error: initializer not constant expression
  static const string m5 = "odd"; // error: not integral type
};
```

C++11 erlaubt:

```
class A { public: int a = 7; } // besser int a(7); noch besser int a{7};
```

als Abkürzung für

```
class A { public: int a; A() : a(7) {} };
```


# inherited constructors

Sie kennen diesen Effekt?:

```
struct B {
 void f(double);
};
struct D : B {
 void f(int);
};
B b; b.f(4.5); // fine
D d; d.f(4.5); // surprise: calls f(int) with argument 4
```


# inherited constructors

In C++98 kann man überladene Funktionen in die Ableitung “hochziehen”, aber keine Konstruktoren ☹

```
struct B {
 void f(double);
};
struct D : B {
 using B::f;  // bring all f()s from B into scope
 void f(int); // add a new f()
};
B b; b.f(4.5); // fine
D d; d.f(4.5); // fine: calls D::f(double) which is B::f(double)
```

# inherited constructors

In C++11 geht das auch für Konstruktoren

```
class Derived : public Base {
public:
 using Base::f; // lift Base's f into Derived's scope
                // -- works in C++98
 void f(char);  // provide a new f
 void f(int);   // prefer this f to Base::f(int)
 using Base::Base; // lift Base constructors Derived's scope
                   // -- C++11 only
 Derived(char);  // provide a new constructor
 Derived(int);   // prefer this constructor to Base::Base(int)
};
```

# lambdas

Generische Algorithmen brauchen oft Hilfsklassen für *functional objects*:

```
class between {
  double low, high;
public:
  between(double l, double u) : low(l), high(u) { }
  bool operator()(const employee& e) {
    return e.salary() >= low && e.salary() < high;
  }
};

double min_salary;
std::find_if(   employees.begin(),
                employees.end(),
                between(min_salary, 1.1*min_salary)
);
```

# lambdas

leider noch nicht
in Xcode :-(

Lambdas sind kleine Funktionen am Ort ihrer Verwendung:

```
double min_salary = ....
....
double u_limit = 1.1 * min_salary;
std::find_if(
        employees.begin(),
        employees.end(),
        [&](const employee& e) {
                return e.salary() >= min_salary && e.salary() < u_limit; }
        );
```

**[&]** ist eine sog. *capture list*, sie gibt an, dass alle lokalen Variablen per Referenz übergeben werden Übergabe alle per Wert: **[=]** Einzelne Variablen können explizit benannt werden **[&x, y]**, Leere *capture list* **[]**,

Der Rückgabetyp wird häufig aus dem *return statement* abgeleitet. Ohne return **void**. Ansonsten ist (nur!) *suffix return type syntax* möglich.

# lambdas

Lambdas können auch benannt und kombiniert werden. Als ‚rufbare' Objekte sind sie mit std::function's kompatibel:

```
#include <functional>
#include <iostream>

std::function<int(int)> twice() {
        return [] (int n) { return 2*n; };
}

std::function<int(int)> addConst(int c) {
        return [c] (int n) { return n+c; };
}

template <typename OP>
int rechne(int n, OP op) {
        return op(n);
}
```

# lambdas

```
std::function<int(int)> operator*
     (std::function<int(int)> f, std::function<int(int)> g) {
         return [f, g] (int n) { return f(g(n)); };
}
int main() {
         auto add1 = addConst(1);
         auto add4 = addConst(4);
         auto dbl = twice();

         int n = 6;
         n = rechne(n, add4);
         n = rechne(n, dbl);
         n = rechne(n, add1);
         n = rechne(n, dbl);
         std::cout << n << std::endl;

         auto h = dbl * add1 * dbl * add4;
         std::cout << h(6) << std::endl;
}
```

# lambdas

Bei der capture list kann man zwischen capture by value und capture by reference wählen:

`[x]` --- x by value (Kopie) (x darf lokal NICHT verändert werden)
`[&x]` --- x by reference

```
[=]   capture all by value (mit Vorsicht verwenden)
[&]   capture all by ref (dito)
```

Mischen ist möglich, aber nach = oder & nur noch die andere Variante aufzählbar, keine Variable darf doppelt vorkommen

Fehlerhaft sind z.B.

```
[=, &x, y]
[&, x, y, &z]
[&, x, y, x]
```

Ersetzungsschema mit
`operator () const` !

# lambdas

Lambdas in Memberfunktionen von Klassen haben keinen Zugang zu den Memberdaten. Sie können diesen aber durch explizites *capture this* erlangen:

```
struct X {
    int some_data;
    void foo(std::vector<int>& vec)
    {
        std::for_each(begin(vec), end(vec),
            [this](int i&) { i += some_data;} );
    }

};
```