# C++0x
# – the next ISO C++

based on

http://www2.research.att.com/~bs/C++0xFAQ.html

by Bjarne Stroustrup

# auto -- deduction of a type from an initializer

```
auto x = 7;
```

**x** ist von Typ **int** wegen des Typs des Literals.

```
auto x = expression;
```

**x** ist vom Typ des Resultats von **expression**.

# auto -- (cont.)

```
template<class T> void printall(const vector<T>& v) {
  for (auto p = v.begin(); p!=v.end(); ++p) cout << *p << "\n";
}
```

statt C++98:

```
template<class T> void printall(const vector<T>& v) {
  for (typename vector<T>::const_iterator p = v.begin(); p!=v.end(); ++p)
    cout << *p << "\n";
}
```

# range for statement

range for erlaubt es, über beliebige Bereiche zu iterieren:

alle Standard Container, std::string, initializer lists, array, (alles was begin() und end() hat).

```
void f(const vector<double>& v) {
  for (auto x : v) cout << x << '\n';
  for (auto& x : v) ++x; //using reference to allow us to change the value
}

for (const auto x : { 1,2,3,5,8,13,21,34 }) cout << x << '\n';
```

**`begin()`** (and **`end()`**) können Member [**`x.begin()`**] oder *free-standing* Functionen sein [**`begin(x)`**].

# defaulted and deleted functions -- control of defaults

Das Idiom “Kopieren verboten” kann nun direkt ausgedrückt werden:

```
class X {
// ...
  X& operator=(const X&) = delete; // Disallow copying
  X(const X&) = delete;
};
```

Auch die Nutzung des *default copy ctors* kann spezifiziert werden:

```
class Y {
  // ...
  Y& operator=(const Y&) = default;
  // default copy semantics
  Y(const Y&) = default;
```

# defaulted and deleted functions -- (cont.)

Auch zur Eliminierung unerwünschter Umwandlungen:

```
struct Z {
// ...
  Z(long long); // can initialize with an long long
  Z(long) = delete; // but not anything less
};
```

# enum class
# -- scoped and strongly typed enums

C - **enum**s mit Problemen:

- konvertierbar nach int
- exportieren ihre Aufzählungsbezeichner in den umgebenden Bereich (name clashes)
- schwach typisiert (z.B. keine forward Deklaration möglich)

**enum class**es ("strong enums") sind stark typisiert und 'scoped':

```
enum Alert { green, yellow, election, red }; // traditional enum
enum class Color { red, blue };              // scoped and strongly typed enum
                             // no export of enumerator names into enclosing scope
                             // no implicit conversion to int
enum class TrafficLight { red, yellow, green };

Alert a = 7;             // error (as ever in C++)
Color c = 7;             // error: no int->Color conversion
int a2 = red;            // ok: Alert->int conversion
int a3 = Alert::red;     // error in C++98; ok in C++0x
```

# enum class -- (cont.)

Typ der Repräsentation kann spezifiziert werden

```
enum class Color : char { red, blue }; // compact representation enum class

TrafficLight { red, yellow, green };
          // by default, the underlying type is int

enum E { E1 = 1, E2 = 2, Ebig = 0xFFFFFFF0U };
// how big is an E?
// (whatever the old rules say;
// i.e. "implementation defined")

enum EE : unsigned long { EE1 = 1, EE2 = 2, EEbig = 0xFFFFFFF0U };
// now we can be specific
```

forward Deklaration möglich

```
enum class Color_code : char; // (forward) declaration
void foobar(Color_code* p); // use of forward declaration
```

# constexpr

## -- generalized and guaranteed constant expressions

mehr Konstanten zur Compile-Zeit

```
enum Flags { good=0, fail=1, bad=2, eof=4 };

constexpr int operator|(Flags f1, Flags f2)
{ return Flags(int(f1)|int(f2)); }

void f(Flags x) {
        switch (x) {
                case bad: /* ... */ break;
                case eof: /* ... */ break;
                case bad|eof: /* ... */ break;
                default: /* ... */ break;
        }
}
```

# constexpr -- (cont.)

```
constexpr int x1 = bad|eof; // ok

void f(Flags f3) {
  constexpr int x2 = bad|f3; // error can't evaluate at compile time
  int x3 = bad|f3; // ok
}
```

Auch für einfache Objekte:

```
struct Point {
  int x,y;
  constexpr Point(int xx, int yy) : x(xx), y(yy) { }
};
constexpr Point origo(0,0);
constexpr int z = origo.x;
constexpr Point a[] = {Point(0,0), Point(1,1), Point(2,2) };
constexpr x = a[1].x; // x becomes 1
```

# decltype

**decltype(E)**

Ist der Typ ("declared type") des Names oder des Ausdrucks **E** und kann in Deklarationen verwendet werden.

```
void f(const vector<int>& a, vector<float>& b) {
  typedef decltype(a[0]*b[0]) Tmp;
  for (int i=0; i<b.size(); ++i) {
    Tmp* p = new Tmp(a[i]*b[i]);
  }
}
```

**auto** ist oft einfacher. **decltype** wird gebraucht, wenn man einen Typ für etwas benötigt, das keine Variable ist z. B. ein return Typ.

# initializer lists

Handliche Listen überall

```
vector<double> v = { 1, 2, 3.456, 99.99 };

list<pair<string,string>> languages = { // parse error in C++98
 {"Nygaard","Simula"}, {"Richards","BCPL"}, {"Ritchie","C"} };

map<vector<string>,vector<int>> years = { // fine in C++0x
  { {"Maurice","Vincent","Wilkes"},{1913,1945,1951,1967,2000} },
  { {"Martin","Ritchards"},{1982,2003,2007} },
  { {"David","John","Wheeler"},{1927,1947,1951,2004} }
};
```

# initializer lists -- (cont.)

Nicht mehr nur für Felder , Argumente vom Typ  std::initializer_list<T> möglich.

```
void f( initializer_list<int> );
f( {1,2} );
f( {23,345,4567,56789} );
f({}); // the empty list
f{1,2}; // error: function call ( ) missing
years.insert({{"Bjarne","Stroustrup"},{1950, 1975, 1985}});

void f(initializer_list<int> args) {
  for (auto p=args.begin(); p!=args.end(); ++p) cout << *p << "\n";
}
```

Konstruktoren mit einem einzigen Argument vom Typ **std::initializer_list** heißen  initializer-list Konstruktoren. Die Standardcontainer, string, regex etc. haben solche.

# 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 --(cont.)

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)} { }
};
```

# 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++ox 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 -- (cont.)

In C++ox 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++0x only
 Derived(char);  // provide a new constructor
 Derived(int);   // prefer this constructor to Base::Base(int)
};
```

# static_assert

Compiler-Assertions (C- assert wirkt zur Laufzeit!)

```
static_assert(constant_expression,string);
```

```
static_assert(sizeof(long) >= 8,
       "64-bit code generation required for this library.");
struct S { X m1; Y m2; }; static_assert(sizeof(S)==sizeof(X)
+sizeof(Y),
       "unexpected padding in S");
```

Nicht zur Prüfung von Laufzeiteigenschaften verwendbar

```
int f(int* p, int n) {
  static_assert(p==0,"p is not null");
  // error: static_assert() expression not a constant expression
}
```

# long long -- a longer integer

Integers mit wenigstens 64 bits

```
long long x = 9223372036854775807LL;
```

kein **`long long long`**

kein **`short long long`**

[insb. **`long`** != **`short long long`**]

# nullptr -- a null pointer literal

**nullptr** ist ein Literal für den Zeigerwert Null, kein integer!

```
char* p = nullptr;
int* q = nullptr;
char* p2 = 0; // 0 still works and p==p2
void f(int);
void f(char*);
...
f(0); // call f(int)
f(nullptr); // call f(char*)
void g(int);
g(nullptr); // error nullptr is not an int
int i = nullptr; // error nullptr is not an int
```

# suffix return type syntax

```
template<class T, class U> ??? mul(T x, U y) { return x*y; }
```

Wie kann man den Type bei ??? angeben? **decltype**?

```
template<class T, class U> decltype(x*y) mul(T x, U y)
                                  // scope problem!
{ return x*y; }
```

Geht nicht, weil **x** und **y** nicht in diesem Bereich liegen ☹

```
template<class T, class U> decltype(*(T*)(0)**(U*)(0))
                        mul(T x, U y) // ugly! and error prone
{ return x*y; }
```

Lösung: Return Typ nach den Argumenten ([] – Notation der Lambdas könnte evtl. auch durch auto ersetzt werden)

```
template<class T, class U> [] mul(T x, U y) -> decltype(x*y)
{return x*y;}
```

# uniform initialization syntax and

C++ hat verschiedene Wege zur Initialisierung, je nach Objekttyp und Kontext.
fehleranfällig und nicht konsistent ☹

```
string a[] = { "foo", " bar" }; // ok: initialize array variable
vector<string> v = { "foo", " bar" }; // error: initializer list for non-aggregate vector
void f(string a[]); f( { "foo", " bar" } ); // syntax error: block as argument
```

und

```
int a = 2;           // assignment style
int[] aa = { 2, 3 }; // assignment style with list
complex z(1,2);      // functional style initialization
x = Ptr(y);          // functional style for conversion/cast/construction
```

und

```
int a(1);   // variable definition
int b();    // function declaration
int b(foo); // variable definition or function declaration
```

# uniform initialization syntax and

C++0x {}-initializer lists für alle Initialisierungen:

```
X x1 = X{1,2};
X x2 = {1,2}; // the = is optional
X x3{1,2};
X* p = new X{1,2};

struct D : X {
 D(int x, int y) :X{x,y} { /* ... */ };
};

struct S {
 int a[3];
 S(int x, int y, int z) :a{x,y,z} { /* ... */ };
 // solution to old problem
};
```

# rvalue references

non-const Referenzen kann man an *lvalues*, const Referenzen an *lvalues* oder *rvalues*,

An non-const *rvalues* kann man keinerlei Referenzen binden . (Damit niemand den Wert von temporären Variablen ändert, die u.U. verschwinden, bevor der Wert benutzt werden kann)

```
void incr(int& a) { ++a; }
int i = 0; incr(i); // i becomes 1
incr(0); // error: 0 in not an lvalue
```

Aber was ist mit:

```
template<class T> swap(T& a, T& b) { // old style swap
  T tmp(a); // now we have two copies of a
  a = b;    // now we have two copies of b
  b = tmp;  // now we have two copies of tmp (aka a)
}
```

# rvalue references -- (cont.)

Wenn Kopieren für **T** teuer ist (z.B. string und vector), wird swap ebenfalls teuer (deshalb gibt es in std spezialisierte swap-Versionen)

In Wahrheit soll am **besten gar nichts** kopiert werden – die Werte von **a**, **b**, und **tmp** sollen nur **bewegt** werden

In C++ox kann man "move constructors" and "move assignments" definieren:

```
template<class T>
class vector { // ...
   vector(const vector&);           // copy constructor
   vector(vector&&);                // move constructor
   vector& operator=(const vector&);// copy assignment
   vector& operator=(vector&&);     // move assignment
};
// note: move constructor and move assignment takes non-const &&
// they can, and usually do, write to their argument
```

&& bezeichnet eine "*rvalue reference*". Eine *rvalue reference* kann man an einen *rvalue* (aber nicht an einen *lvalue*) binden

# rvalue references -- (cont.)

```
X a;
X f();
X& r1 = a;        // bind r1 to a (an lvalue)
X& r2 = f();      // error: f() is an rvalue; can't bind
X&& rr1 = f();    // fine: bind rr1 to temporary
X&& rr2 = a;      // error: bind a is an lvalue

template<class T> void swap(T& a, T& b) { // "perfect swap" (almost)
  T tmp = move(a);        // could invalidate a
  a = move(b);            // could invalidate b
  b = move(tmp);                  // could invalidate tmp
}
```

**move(x)** bedeutet “man kann **x** als *rvalue* verwenden".

**move()** wiederum kann als template function mit einem rvalue reference Parameter implementiert werden.

*rvalue references* können auch für *perfect forwarding* benutzt werden.

In der C++ox standard library haben alle Container *move constructors* und *move assignments*. Operationen, die Elemente einfügen (wie **insert()** und **push_back()** ) haben Versionen, die auf *rvalue references* arbeiten

# rvalue references -- (cont.)

```
#include <vector>

class X  {
   std::vector<double> data;
public:
   X():  data(100000) {} // lots of data

   X(X const& other): // copy constructor
      data(other.data) {}   // duplicate all that data

   X(X&& other):  // move constructor
      data(std::move(other.data)) {}  // move the data: no copies

   X& operator=(X const& other) { // copy-assignment
      data=other.data; // copy all the data
      return *this;  }

  X& operator=(X && other) { // move-assignment
      data=std::move(other.data); // move the data: no copies
      return *this;   }
};
```

# rvalue references -- (cont.)

```
X make_x()// build an X with some data
{ return X(); }

int main()
{
   X x1;
   X x2(x1);                  // copy
   X x3(std::move(x1)); // move: x1 no longer has any data

   x1=x2;           // copy assign
   x1=make_x(); // return value is an rvalue, so move rather than copy
}
```

# PODs (generalized)

C++98 beschränkt POD (plain old data) auf C-structs, obwohl vieles aus Klassen am Layout nichts ändert.

```
struct S { int a; }; // S is a POD
struct SS { int a; SS(int aa) : a(aa) { } };
// SS is not a (C++98-)POD
struct SSS { virtual void f(); /* ... */ };
```

In C++0x, S und SS sind "standard layout types" (a.k.a. POD). SSS ist auch kein POD wegen dem vptr.

PODs sind trivial kopierbare Typen, triviale Typen, und standard-layout Typen. POD ist rekursiv definiert: Wenn alle Member POD's sind und die Klassen keine virt. Funktionen, virtual Bases oder Referenzen enthält, ist sie selbst POD

# raw string literals

Backslash-Escape-Horror (insb. bei regulären Ausdrücken: Muster Wort"\"Wort )

```
string s = "\\w\\\\\\w"; // I hope I got that right
```

In einem "raw string literal" ist der backslash einfach ein backslash

```
string s = R"[\w\\\w]"; // I'm pretty sure I got that right
```

Warum nicht R"...." ? Häufig braucht man " im quoted string!

```
R"["quoted string"]" // the string is "quoted string"
```

Was ist mit **]** in einem a raw string? Eher selten und **"[...]"** ist nur das *default delimiter pair*.
Beliebige Zeichen vor **[ und** nach **]** sind erlaubt.

```
R"***["quoted string containing the usual terminator ("])"]***"
// the string is "quoted string containing the usual terminator ("])"
```

# user-defined literals

Nutzerdefinierte Typen und built-in Typen sind gleichberechtigt, außer dass es keine Literale dieser Typen gibt ☹

```
"Hi!"s // string, not ``zero-terminated array of char''
1.2i // imaginary
123.4567891234df // decimal floating point (IBM)
101010111000101b // binary
123s // seconds
123.56km // not miles! (units)
12345678901234567890123456789012345678901234567890x // extended-precision
```

In C++ox sind *user-defined literals* möglich durch Definition sog. *literal operators*.

```
constexpr complex<double> operator "" i(long double d) // imaginary literal
{ return {0,d}; } // complex is a literal type

std::string operator""s (const char* p, size_t n) // std::string literal
{ return string(p,n); } // requires free store allocation
```

**constexpr** erlaubt statische Auswertung z.B. `1+2i`

# user-defined literals -- (cont.)

Argumente können “cooked” – als Literal des Typs ohne Suffix oder aber “uncooked” – als String übergeben werden.

Es gibt 4 Arten von user-defined literals:

- integer literal: Literaloperator mit einem Argument vom Typ **unsigned long long** oder **const char*** .
- floating-point literal: Literaloperator mit einem Argument vom Typ **long double** oder **const char*** .
- string literal: Literaloperator mit zwei Argumenten der Typen **(const char*, size_t)** ! 2. Argument darf nicht fehlen: Wenn man eine ‘andere Sorte Strings’ haben will, muss die Länge immer bekannt sein!
- character literal: Literaloperator mit einem Argument vom Typ **char** .

# user-defined literals -- (cont.)

```
Bignum operator"" x(const char* p) { return Bignum(p); }
void f(Bignum);
f(1234567890123456789012345678901234567890x);
// C-style string "1234567890123456789012345678901234567890"
// is passed to operator"" x().
```

Suffixe sind (gewollt) kurz – Namespaces vermeiden *name clashes*:

```
namespace Numerics { // ...
 class Bignum { /* ... */ };
 namespace literals { operator"" X(char const*); }
}
using namespace Numerics::literals;
```

# attributes

Eine standardisierte Syntax für alle Arten optionaler und Vendor-Spezifischer Informationen im Quelltext (z.B. **__attribute__**, **__declspec**, **#pragma** ...), fast überall erlaubt, betreffen immer das vorhergehende Element.

```
void f  [[ noreturn ]] () { throw "error"; /* OK */ }
// f() will never return
unsigned char c [[ align(double) ]] [sizeof(double)];
// array of characters, suitably aligned for a double
```

Neben **noreturn** und **align** gibt's im Standard noch:

```
struct B { virtual void f [[ final ]] (); // do not try to override };
struct D : B { void f(); // error };
struct foo* f [[carries_dependency]] (int i); // hint to optimizer
int* g(int* x, int* y [[carries_dependency]] );
```

Geplant ist z.B. OpenMP-Unterstützung

```
for [[omp::parallel()]] (int i=0; i<v.size(); ++i) { ... }
```

# 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 -- (cont.)

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 Vartiablen können explizit benannt werden **[&x, =y]**, Leere *capture list* **[]**,

Der Rückgabetyp wird häufig aus dem *return statement* abgeleiter. Ohne return **void**. Ansonsten ist *suffix return type syntax* möglich.

# inline namespace

Koexistenz verschiedener Bibliotheksversionen . Insb. für **::std** benötigt

```
// file V99.h:
inline namespace V99 {          void f(int); // does something better than the V98 version
                                void f(double); // new feature
}
// file V98.h:
namespace V98 {                 void f(int); // does something
}
// file Mine.h:
namespace Mine {
#include "V99.h"
#include "V98.h"
}
```

Beide Versionen sind verfügbar: Deklarationen aus inline Namensräumen erscheinen im übergeordneten Namensraum

```
#include "Mine.h"
using namespace Mine;
V98::f(1); // old version
V99::f(1); // new version
f(1); // default version   --- ??? vermutlich V98::f(1)
```

# explicit conversion operators

**explicit** auch für *conversion operators*

```
struct S { S(int) { } };
struct SS {
  int m;
  SS(int x) :m(x) { }
  explicit operator S() { return S(m); }
  // because S don't have S(SS)
};
SS ss(1);
S s1 = ss; // error; like an explicit constructor
S s2(ss);  // ok ; like an explicit constructor
void f(S);
f(ss);     // error; like an explicit constructor
f(S(ss));  // ok
```

# template related stuff ...

- right-angle brackets (>>)
- template alias (formerly known as "template typedef")
- variadic templates
- local types as template arguments
- extern templates
- [ concepts, concept maps and axioms nicht in C++ox ]

# ::std:: related stuff ...

- algorithms improvements
- container improvements
- scoped allocators
- unordered containers
- std::array
- std::forward_list
- std::tuple
- std::function and std::bind
- std::unique_ptr
- std::shared_ptr
- std::weak_ptr

zum Teil schon in `std::tr1`

# thread related stuff ...

- memory model
- threads
- mutual exclusion
- locks
- condition variables
- time utilities
- atomics
- std::future and std::promise
- std::async()
- abandoning a process

# other stuff ...

- unions (generalized)
- random number generation
- regular expressions

zum Teil schon in `std::tr1`